
Universidade de Aveiro
2009
Departamento de Electrónica, Telecomunicações e Informática

Rui Miguel Soares Gomes

PERIFÉRICOS INTELIGENTES COM BARRAMENTO DIGITAL DE COMUNICAÇÃO

INTELLIGENT PERIPHERALS WITH DIGITAL COMMUNICATION BUS

Universidade de Aveiro
2009
Departamento de Electrónica, Telecomunicações e Informática

Rui Miguel Soares Gomes

PERIFÉRICOS INTELIGENTES COM BARRAMENTO DIGITAL DE COMUNICAÇÃO

INTELLIGENT PERIPHERALS WITH DIGITAL COMMUNICATION BUS

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia Electrónica e Telecomunicações (M.I.E.E.T.), realizada sob a orientação científica do Prof. Doutor Manuel Bernardo Salvador Cunha, Professor Auxiliar do Departamento de Electrónica Telecomunicações e Informática da Universidade de Aveiro

Apoio financeiro da FCT e do FSE no âmbito do III Quadro Comunitário de Apoio.

O júri

Presidente	Prof. Doutor António Manuel de Brito Ferrari Almeida Professor Catedrático da Universidade de Aveiro
Arguente	Prof. Doutor Jorge Miguel Nunes dos Santos Cabral Professor Auxiliar no Departamento de Electrónica Industrial da Universidade do Minho
Orientador	Prof. Doutor Manuel Bernardo Salvador Cunha Professor Auxiliar no Departamento do DETI
Co-Orientador	Prof. Doutor José Luís Costa Pinto de Azevedo Professor Auxiliar no Departamento do DETI

Agradecimentos

Dedico esta dissertação à minha família a amigos pelo apoio incessável durante o decorrer do projecto. Dedico também ao Eng. Pedro Kulzer pelo incansável apoio e optimismo, aos meus colegas pelo espírito de equipa e ao Prof. Doutor Bernardo Cunha pelo contínuo apoio e pelo contributo intelectual ao projecto.

Palavras-chave

Unidade de Controlo Electrónico, ECU, Motor, Periféricos Inteligentes, Sensor, Actuador, Barramento Digital, Comunicação Série, Diagnóstico.

Resumo

Esta dissertação é parte integral do projecto ECU2010 e é focada no desenvolvimento de Periféricos Inteligentes que são conectados à ECU através de um Barramento Digital de Comunicação.
O projecto ECU2010 está centrado no desenvolvimento de uma nova arquitectura da unidade de controlo electrónico (ECU) para desporto automóvel, focada no controlo de motores de combustão interna.
A arquitectura proposta deverá ser ter a capacidade de controlar um motor de combustão interna usando os mais modernos modelos de controlo, mas sendo baseada numa modelo de processamento distribuído, composta por módulos de processamento auto-suficientes ao nível de comunicações e armazenamento e de sensores/actuadores com inteligência capazes de processamento prévio de dados.
A ênfase desta dissertação será colocada apenas nos Periféricos Inteligentes e no Barramento Digital de Comunicação.
Este documento irá analisar e propor uma solução para a inclusão de capacidades de processamento, armazenamento e diagnóstico nos periféricos, assim como o desenvolvimento de um Barramento Digital de Comunicação que permite aos periféricos comunicarem com a ECU e sincronizarem-se com a rotação do motor.

Keywords

Electronic Control Unit, ECU, Engine, Intelligent Peripherals, Sensor, Actuator, Digital Bus, Serial Communication, Diagnostic.

Abstract

This dissertation is an integral part of the ECU2010 project and is focused on the development of Intelligent Peripherals which connect to the ECU by means of a Digital Communication Bus.
The ECU2010 project is centered on developing a new architecture of electronic control units (ECU) for motor sport, focussing on control of internal combustion engines.
The proposed new architecture should be capable of controlling an internal combustion engine using the state-of-the art control models, but based on a distributed processing model consisting on self-sufficient processing modules in terms of communications, storage and intelligent enabled sensors/actuators, which is able to produce low-level data processing.
The focus of this dissertation will only be the Intelligent Peripherals and the Digital Communication Bus.
This document will analyse and propose a solution for the incorporation of processing, storage and diagnostic capabilities into peripherals, as well as the development of a Digital Communication Bus which allows the peripherals to communicate with the ECU and synchronize them with the engine's rotation.

Table of contents

Table of figures

Acronyms and abbreviations

ACK	-	Acknowledge
ADC	-	Analog-to-Digital Converter
ALU	-	Arithmetic Logic Unit
BCDFP	-	Binary Coded Decimal Floating Point
CMOS	-	Complementary Metal Oxide Semiconductor
CPU	-	Central Processing Unit
CRC	-	Cyclic Redundancy Check
DAC	-	Digital-to-Analog Converter
DMA	-	Direct Memory Access
ECU	-	Electronic Control Unit
EOF	-	End Of Frame
FPGA	-	Field-Programmable Gate Array
GIMy	-	Gateway Intelligent Memory
IDMS	-	Integrated Development and Management System
IC	-	Integrated Circuit
I/O	-	Input/Output
I2C	-	Inter-Intergrated Circuit
JTAG	-	Joint Test Action Group
MOSFET	-	Metal Oxide Semiconductor Field Effect Transistor
NMOS	-	N-channel MOSFET
NRZ	-	Non Return to Zero
NTC	-	Negative Temperature Coefficient
PMOS	-	P-channel MOSFET
RAM	-	Random Access Memory
RISC	-	Reduced Instruction Set Computing
RPM	-	Rotations-Per-Minute
SOF	-	Start Of Frame
SPI	-	Serial Peripheral Interface
SSI	-	Synchro-Serial Interface
STP	-	Shielded Twisted Pair
TDC	-	Top Dead Centre
TQFP	-	Thin Quad Flat Pack
TTL	-	Transistor-Transistor Logic
USART	-	Universal Synchronous Asynchronous Receiver Transmitter
USB	-	Universal Serial Bus

UTP	-	Un-shielded Twisted Pair
VHDL	-	VHSIC Hardware Description Language
VHSIC	-	Very High Speed Integrated Circuits

Chapter 1
Introduction

Summary

This chapter begins by defining the thesis guidelines, followed by motivation and an overview of the objectives. It is completed by a brief description of the thesis organization.

1.1. Guidelines

The automotive industry is continuously evolving. Since the first internal combustion engine propelled vehicle, the world has witnessed huge advances in the technologies applied in an automobile.

Innovation should also be applied in the engine control systems, that have suffered little changes other than increasing the speed, memory and, sometimes, complexity. Also, the number of peripherals is constantly increasing. This requires an ever growing ECU in terms of peripheral interfaces and processing power to cope with that increase and new functionalities.

This dissertation is part of a project called ECU2010 which is product of a joint venture between *Aveiro University*, *Kulzer Consultores Técnicos* and *Bosch Motorsports*. Its goal is to develop a new paradigm in terms of ECU conception. Ease of use, fast system configuration, scalability and faster problem detection and debugging are some of the main objectives. The system is composed by an ECU connected to several intelligent peripherals through a digital communication bus. The ECU has a modular architecture were modules can be added or removed when needed. Each module has custom processing capabilities and a peripheral bus associated. Each peripheral has the same processing capabilities as the modules. The ECU has USB and wireless connectivity. An integrated development and management application is responsible for the system programming and configuration.

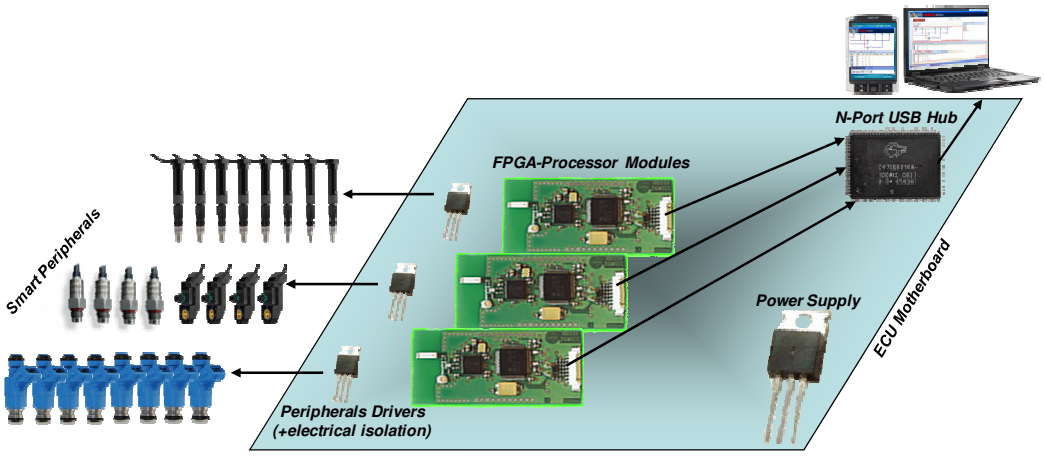

Figure 1 – Conceptual system overview.

The focus of this part of the project is to develop a new peripheral design approach, involving peripherals with local intelligence, via an incorporated processor, connected to the ECU by a digital bus.

1.2. Motivation

Despite all the new technologies, some aspects have changed very little through the years in the engine control systems. Analog technology has been the standard for peripheral interface since the beginning of electronic engine control. Digital interfaces haven't been very appealing to the mass production. It poses some problems to its implementation mainly in terms of cost effectiveness of the peripheral itself as of the cabling that connected it to the ECU witch in large quantities becomes a disadvantage in comparison with the analog technology. But with the advances in technology which can produce smaller, faster and more reliable ICs and the more demanding market in terms of safety, performance and maintenance make the digital counterpart become the logical path for the future.

Meanwhile, for the Motorsports segment this is a viable option. There, system cost is not a major problem and there is a large demand for custom solutions that provide fast workaround times and extra capabilities.

Delegating part of the processing and the power drivers to the peripherals brings many advantages and many new possibilities, from lifetime counters built in each peripheral to a completely digital ECU.

1.3. Objectives

Prove the concept that a new digital solution, involving local peripheral intelligence with digital communication with the ECU, is a valid and, in some cases, better approach into solving problems that exist in nowadays ECUs.

1.4. Organization

This dissertation is organized in six separate parts.

The first part, or chapter 1, is an introduction where the objectives, the motivations and the main guidelines are presented.

The second part describes the state of the art, similar existing technologies to those that are being presented in this dissertation.

Chapter 3, or the third part, is where the developed work is described in detail. It includes the Digital Communication Bus, the Intelligent Peripherals and also the software component.

In the fourth part the results are mentioned and discussed. Then, in chapter 5, the conclusions reached are presented.

The last part includes the references found throughout this document and also the bibliography. The main parts of the code are included in the attachments and are considered confidential so they may not be distributed to third parties.

Chapter 2
Analysis of possible solutions

Summary

In this chapter it will be given an overview of some of the existing technologies. Also, the chosen solution will be proposed.

2.1. Digital Bus

A wide range of digital buses are currently available in the market. For automotive industry some are focused on entertaining areas, others on comfort and others on chassis.

2.1.1. CAN Bus Description

The CAN bus protocol [1] was officially released in 1986 by Robert Bosch GmbH. The CAN bus has been adopted by several automotive brands as a vehicle diagnostic interface bus and also as interface for some peripherals. Electrically it is based on a differential 2-wire interface, usually Shielded Twisted Pair (STP), Un-shielded Twisted Pair (UTP) or Ribbon cable. It uses Non Return to Zero (NRZ) bit encoding and bit-stuffing, providing compact messages and high immunity to external interference. Data rates from 10kbps (the minimum rate) up to 1Mbps are possible even though all modules must support 20kbps for protocol compatibility. Normally, all nodes on a specific bus use the same data rate.

The maximum cable length depends on the data rate used varying from a maximum line length of 1Km at 50Kbps to 40 meters at 1Mbps. Termination resistors are used at each end of the cable, and the recommended value is 120 ohm.

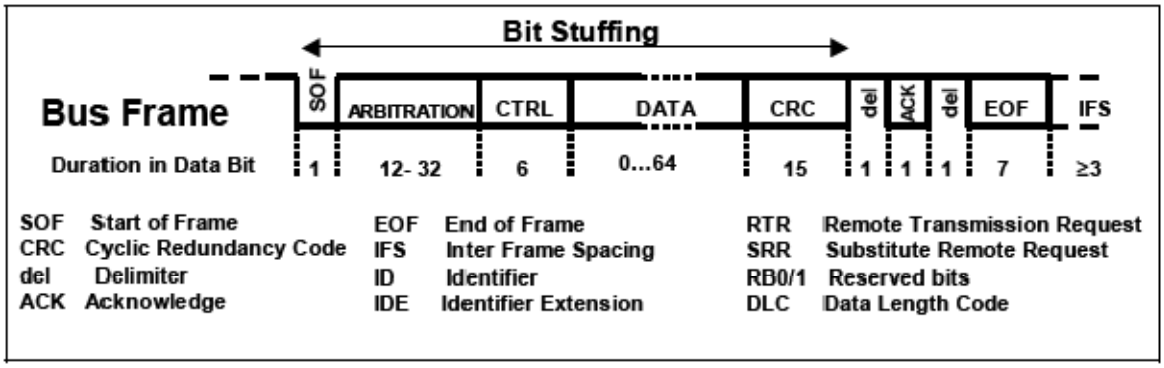

Figure 2 – CAN Bus frame topology.

The CAN bus interface uses asynchronous data transfer. The data frame is composed of several fields, respectively: Arbitration, Control, Data, CRC and ACK. The frame begins with a 'Start of frame' (SOF), and ends with an 'End of frame' (*EOF*) space. The data field may contain from 0 to 8 bytes. The frame error check is based on a 15 bit Cyclic Redundancy Code (CRC). CAN implements five error detection mechanisms, 3 at the message level (CRC, Frame Checks, and Acknowledgment Error Checks) and 2 at the bit level (Bit Monitoring and Bit Stuffing).

Its main drawbacks are the software configuration, which can prove to be somewhat complex as well as electrical problem debugging.

2.1.2. 'byteflight' Bus Description

The 'byteflight' [2] bus is a new protocol developed by BMW and some semiconductor companies. It is a combination of time and priority controlled bus access. It combines the advantages of synchronous and asynchronous protocols. Information is sent in frames but different levels of priority can be set. Each frame is composed by the ID field, the frame Length, the Data (0 to 12 bytes) and the CRC field (15 bits). Maximum bus data rate is 10Mbit/s providing a minimum of 5Mbit/s at full bus load (optical fibber). Optical fibber is used to reduce Electromagnetic Interference (EMI) but electrical transceivers can be used at lower bitrates. NRZ coding is used between the controller and the transceiver chip. The protocol provides collision free communication and a deterministic behaviour.

Possible bus configurations are star, bus and cluster. Already implemented is a star topology with bidirectional (half-duplex) communication using optical fibber. Some bus diagnostic functions are available such as optical transmission quality.

This is a proprietary bus so the access to information, components and software is rare. Also, it is a recent protocol there may exist problems that haven´t been solved or even found.

2.1.3. LIN Bus Description

The Local Interconnect Network (LIN) [3] bus was created in 1998 and is widely used in the automotive area. It is a serial communication bus used to interconnect intelligent sensors and actuators. The maximum communication speed on a LIN bus is 19200 baud over a single wire. Baud rates of 2400 and 9600 are also possible. At a maximum bus speed of 19200 baud, the maximum cable length is 40 meters. LIN nodes can send or receive 8 byte commands every 10ms, 5ms for shorter commands. It also uses a Master/Slave configuration. There are no bus collisions because only one message is allowed on the bus at a time. All communications are initiated by the Master.

The LIN message protocol is composed by the Master request and the Slave response. The Master Block is composed by a “Sync Break” field with 13 bits used to identify the start of the frame, a “Sync” field to allow the Slave to synchronize and a message “Identifier” field with 8 bits. The Slave response is composed by 1 to 8 bytes of data and an 8 bit CRC block.

The single wire interface and the low cost are attractive characteristics but the low speeds available make it a difficult option for engine control.

2.1.4. MI bus description

The Motorola Interconnect (MI) [4] bus is a serial communications interface that uses a single wire. It is similar to LIN and also uses a Master/Slave topology. Despite the simplicity of the one wire bus, its low speed limits its use to comfort functions.

2.2. Intelligent Peripherals

Intelligent Peripherals or even “smart peripherals” are still not massively used. There are still few manufacturers that produce such specific products. Mainly because there is still no large requests for smart peripherals and they are still somewhat recent technology. There are very few products in the market that resemble the concept that is going to be described in this dissertation. So, only a few related examples will be presented here.

2.2.1. 3D accelerometer

This is a 3 axis accelerometer from Nomadic Solutions [5] with RS232 interface. This sensor has an internal controller that reads the acceleration sensor, converts them into ASCII frames and sends them through the RS232 interface at a configurable rate. The

newly read values for each axis are transmitted as an average of the last 3 measures. Thresholds can be defined to condition the transmission of values.

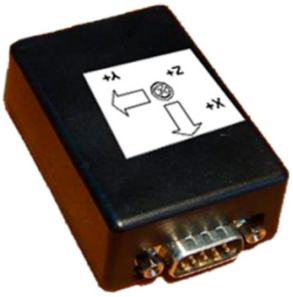

Figure 3 - 3D Accelerometer from Nomadic Solutions.

2.2.2. TinyTIM™ (Tiny Transducer Interface Module)

The TinyTIM module is a Wireless Bluetooth Smart Sensor Module from Smart Sensor Systems [6]. Besides the wireless connectivity, this module incorporates a 37 pin plug providing several functionalities. Among them, digital I/O, 4 channel 16 bit ADC, 2 channel 12 bit ADC and a SPI interface. It has all the signal conditioning circuits and an internal controller that converts all measures into their corresponding physical values removing this processing work from the destination processor.

It also includes an interesting feature. All of the necessary information such as the datasheet is included in the sensors memory and can be downloaded at any time. This eliminates the need of additional media or resources to provide the information needed to operate the module.

Figure 4 – TinyTIM Module.

Chapter 3
Description

Summary

In this chapter it will be described how the project was implemented and the various components will be explained in detail.

3.1. Digital Bus

3.1.1. Overview

The various peripherals in the car, sensors and actuators, need to send and receive information to and from the ECU. This communication relies on a bus that has to obey to certain rules and parameters in order to be as efficient as possible, such as:

- high data rate;
- high noise immunity;
- low hardware complexity;
- low software overhead;
- low error rate;
- low hardware cost;

Another important aspect is the cabling. In regular passive sensors, for example, a resistive temperature sensor, it would only need two wires to interface it to the ECU through an analog bus, normally Ground and Signal. Also, as the bandwidth needed for this analog signal is very low, the cable doesn't need to be a coaxial or a shielded cable which are more expensive than the regular two stranded wires. So, the digital bus should, ideally, also have only two wires.

This poses a problem, how to send data and power using only two wires. A possible solution is to have a higher voltage on the bus than the peripheral needs and use the positive voltage periods to supply it, having a capacitor to sustain the power during the periods when the bus has no voltage.

3.1.2. Protocol

Protocol highlights:

- Time coded bits.
- Asynchronous protocol.
- Master-Slave architecture.
- Plug and Run type operation.
- Up to 16 Slaves simultaneously.
- Message format: SYNC, CMD, PLUG ID, MSG ID, DATA, CRC.
- Collision free communication, Master initiated communication.
- Data rate: 2 Mbit/s max.
- Deterministic behaviour ensured by protocol.

The information is binary encoded. The bits are encoded using different timings in the transitions of the bus state. The ‘0’ bit corresponds to a short interval between transitions, and the ‘1’ bit to a long interval. This avoids the possibility of the bus being for long periods of time in a state where it has no voltage, thus not delivering power to the peripherals. For a good compromise between higher bit rate, simpler VHDL code and good error margins, the chosen ratio between the periods of either bit is 1:2. That is to say that the ‘1’ bit is twice as long as the ‘0’ bit. Lowering the ratio would allow for a higher bit rate but would worsen the error margins and vice-versa.

It is possible to have different bit rate messages on the bus because each frame has a synchronization field in the beginning that allows the receiver to decode the frame correctly. This allows the bus devices to run at different speeds, as long as they are slower than the device on the ECU side (Bus Master) but also not too slow that the internal counters overflow. These limits can be adjusted for the specific application.

Each communication is initiated by the Bus Master (ECU). If the message is a write command, there is no response from the addressed Bus Slave (Peripheral). If on the other hand, the message is a read command, the addressed Bus Slave responds with the requested data after it has been addressed by the Bus Master. In case the Bus Slave does not respond in time, the Bus Master detects that a timeout has occurred and jumps to the next transmission.

Any Peripheral can be plugged and unplugged at any time. There can be up to 16 peripherals connected in a single bus. Each plug has its own identifier which is composed by a simple resistor. The value of this resistor is read by the Peripheral at time of connection and each 2 seconds. The value is translated to an Identifier (ID) that becomes the Peripheral’s address in the bus. After validating its own ID, the Peripheral sends a message to the ECU identifying himself on the bus. A group of messages are then sent to

describe the name, the type, the internal diagnostics, etc. This solution was adopted for its simplicity, low cost and effectiveness.

An intrinsic characteristic of this approach is the possibility of having a non-sequential plug identifier order, if desired.

The “Plug and Run” type operation can be enabled or disabled in the IDMS (Integrated Development and Management System). This feature is embedded in the IDMS because there weren’t enough resources available in the FPGA to implement it in hardware, although it would be possible. This feature is described with more detail in section 3.3.3.

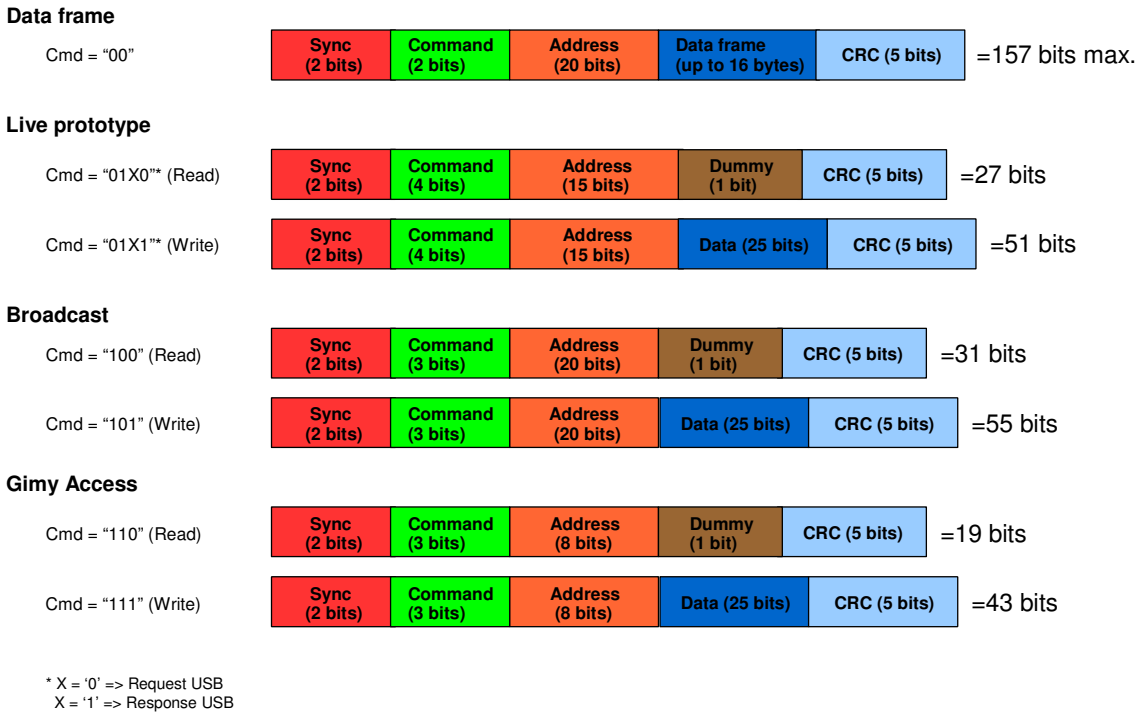

Figure 5 - Data Frame Topology

All messages start with a synchronization field composed of a ’0’ bit and a ‘1’ bit. This field allows the receiver to acquire the timing for each bit. Following the synchronization, the command field is transmitted which has variable length. It may vary from 2 to 4 bits, depending on the message type. The first two bits define the message type, the next one defines whether it is an USB message request or response, and the last one (3rd or 4th bit depending on the command size) defines whether it is a read or a write command.

After the command field, comes the address field with 3 different lengths, 8, 15 and 20 bits, depending on the message type. In 8 and 15 bit lengths, the first 4 bits contain the plug address. These 4 bits enable the possibility to address 16 different devices on the bus. In the 20 bit length, the plug address is composed by 16 bits. This time the address field does not address only one but multiple peripherals, where each bit corresponds to a specific peripheral. If a certain bit is set, it means that the message is destined to the peripheral addressed by that bits position in the field. On the address with a length of 8 bits, the latter 4 bits have the destination peripherals plug address. In the case when it is composed by 15 bits, the last 11 bits contain the direct Gateway Intelligent Memory (GIMy) as this mode is used to read values directly from the memory. In the last case, the remaining 16 bits are used for message broadcast where each bit corresponds to a different plug on the bus.

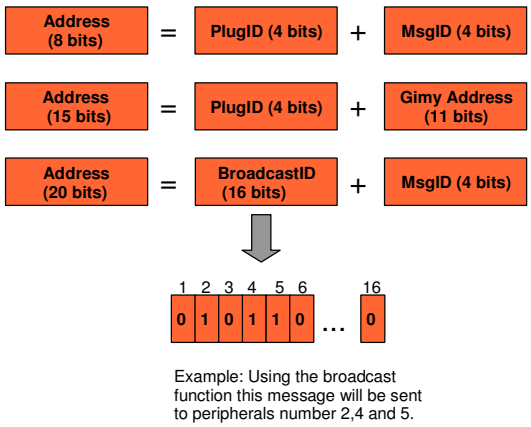

Figure 6 - Available addressing modes.

The following field can be either the data itself or a dummy bit. Except for the case when a data frame is sent and where up to 16 bytes can be sent, the data field is composed by 25 bits that contain a Binary Coded Decimal Floating Point (BCDFP) value. This encoding is part of the systems architecture and is used to represent every value in the system. The dummy bit is used to keep the number of bits in an odd count. This is due to the fact that the bus must return to its idle state (high) and to do that it needs an even number of transitions. As the first transition do not contain information, an odd number of bits correspond to an even number of transitions.

At the end of the frame, the CRC block contains the 5 bit CRC polynomial of the entire frame. The polynomial used was calculated specifically for this application to provide a more efficient code in terms o FPGA resources. The chosen polynomial is x5+x3+x2+x1+1 which is irreducible, providing better error detection, and requires lower number of gates for its implementation.

3.1.3. Bus Topology and Physical Layer

Physical Layer highlights:

- Serial communication.
- Bus-Multi Drop topology.
- Plug identifier.
- Hot swap capabilities.
- Power over bus (low power sensors).
- 6 to 18 Volts supply operation.

The bus has a serial configuration to reduce the wire count. There's a Data line and a Synchronization line, both with a ground connection.

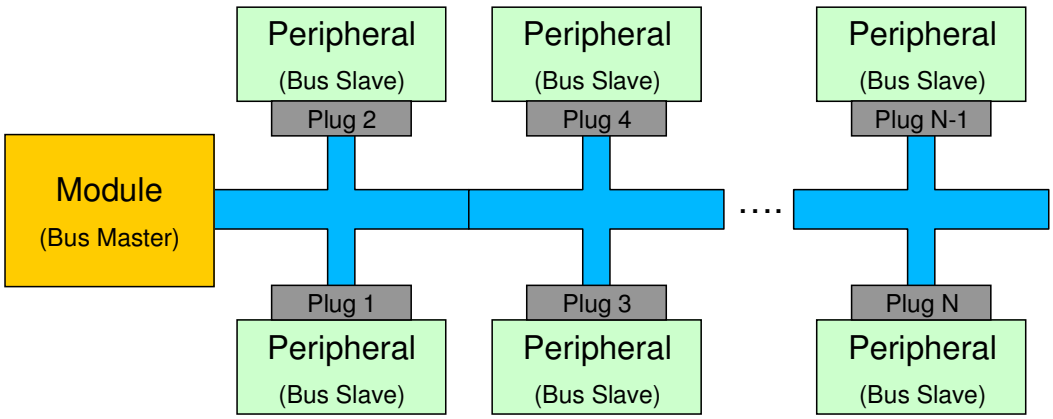

Figure 7 – Digital Bus topology.

The Data line provides the channel for the exchange of information between the devices on the bus. The information is encapsulated in message frames as illustrated in Figure 5.

The Synchronization line is responsible for delivering the current position of the engine to all peripherals connected in the bus. This synchronization is a raw stream of pulses. Each pulse corresponds to a rotation of about one degree of the engine's crankshaft. To synchronize each rotation, a larger pulse is sent each time the crankshaft completes a full cycle (two rotations or 720 degrees). Each peripheral decodes this pulse and checks if it has received the right number of pulses. If this number is not exactly

correct it detects that an error has occurred and takes the necessary actions to avoid engine destruction.

For low power sensors such as temperature sensors or pressure sensors, the power to supply the peripheral can be drawn from the bus. A simple circuit with a diode and a capacitor is used to collect the voltage from the bus. This characteristic allows the bus to have fewer wires, eliminating the supply conductor.

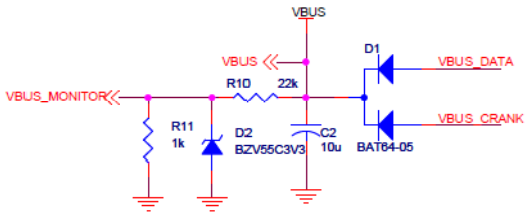

Figure 8 - Bus voltage rectifier and voltage monitor interface.

There are situations when the voltage that supplies the ECU may vary to a great degree. For example, in an everyday situation, when turning on the engine starter, battery voltage can drop down to 6 Volts or even less. On the other hand, when the battery is disconnected while the engine is running and in this case, battery voltage can rise up to 18 Volts. Still, the ECU must work in every case. To do so, the bus has a power supply module which boosts the voltage to 18 Volts and that becomes the reference voltage. This allows the bus to work properly whatever voltage the battery has. The implemented circuit is based on the one found on the datasheet of the IC used. The resistance R2 was added to eliminate lock-up conditions during power up and R3/R4 were chosen to set the output voltage to the desired 18V.

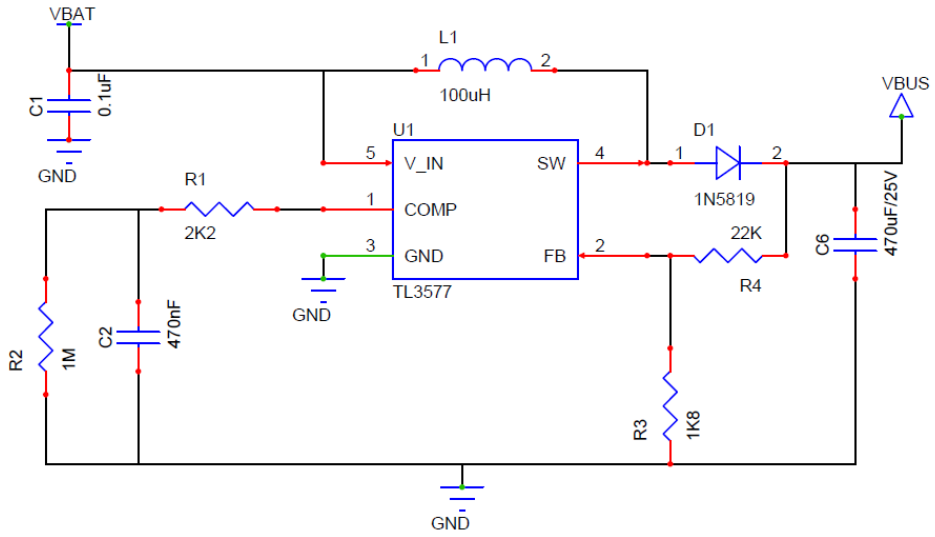

Figure 9 - Bus power supply schematic.

3.1.4. Bus Drivers

On the Bus Master side, the driver is composed by a MOSFET driver IC, TC4431 from Microchip. It receives the signals in TTL 3.3Volts and drives the MOSFET gates directly. It allows lower component count, provides faster transition between states on high capacitive loads (like high power MOSFET gates) driving 1000 pF in 25nsec and has high current drive capability with 1.5A maximum drive current.

The power MOSFET is IRF7309. In a single IC it contains two complementary MOSFETs (P and N type) each with a minimum RDS(on) of 0.05Ohm and 0.1Ohm respectively and a maximum VDS voltage of 30Volts(-30Volts for PMOS). A sensing resistor is connected in series with the PMOS to allow the bus voltage to go low when the bus is at the idle state (PMOS on). This resistor should be 5 Watt power rating as it dissipates an average of 2.89 Watts at high bus communication loads.

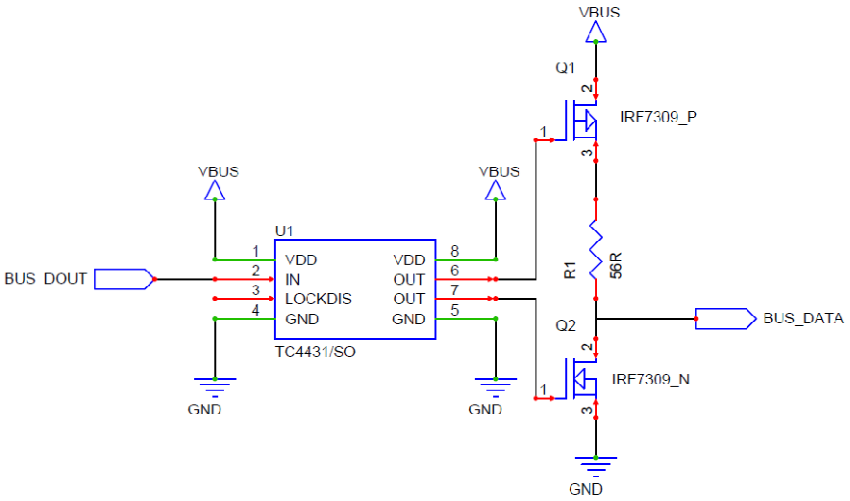

Figure 10 - Bus line driver schematic (*Master*).

On the Bus Slave side, the circuit is identical only lacking the positive drive MOSFET and the sensing resistor. The remaining PMOS on the IRF7309 IC is used on the decoding circuit maximizing component re-usage.

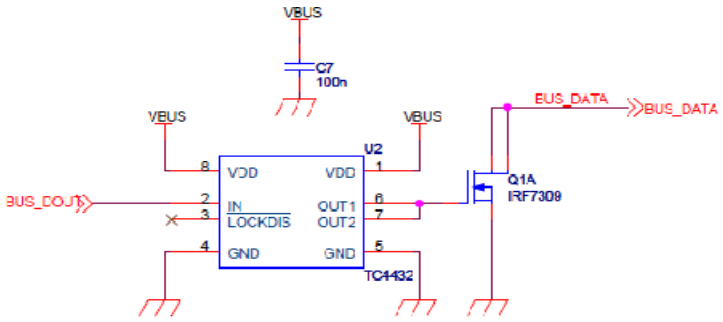

Figure 11 - Bus line driver (*Slave*).

The data line decoding circuit is illustrated in Figure 12. Due to the fact that the cable impedance can reach several ohms depending on cable types, when a Bus Slave drives the bus to a low state, the voltage in the Bus Master side won't reach the ideal zero volts. So, a PMOS (Q1) was connected to the data line to ensure that even a weak pull-down (3 volts drop) from the Bus Slave is detected. The signal is then attenuated so that Q2 turns off faster.

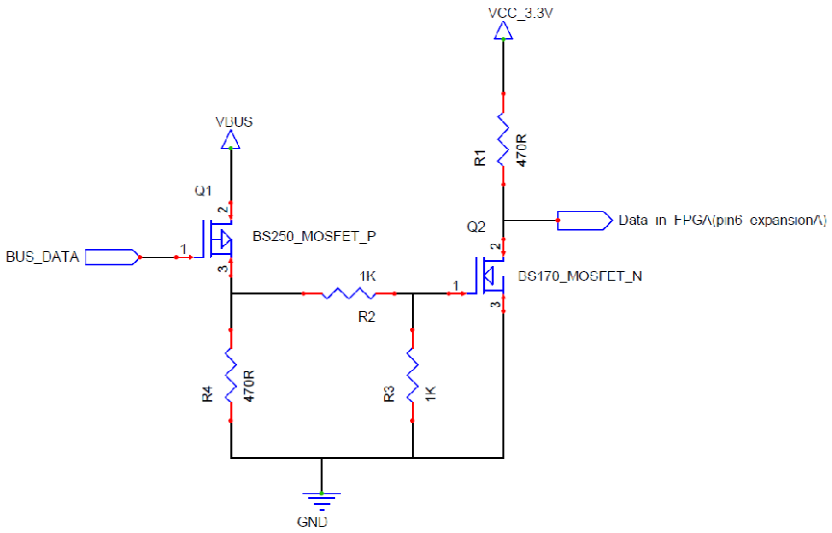

Figure 12 - Bus line decoder (Master).

Pull up/down resistors are used instead of CMOS architecture to reduce power consumption during transitions and to reduce the complexity and cost of the circuit.

Figure 13 illustrates the Bus Slave decoder circuit. The inductor L1 absorbs high frequency spikes in the transitions of the signal in order to avoid false triggers of Q3. R2 and R3 form a voltage divider that allows for Q3 to turn off at higher bus voltages reducing $t_{dON-OFF}$. The output from Q3 connects directly to the FPGA to be processed but, in order to be able to deliver bootstrap sequences to the microcontroller, the signal needs to be inverted. To invert the signal, it is used the remaining PMOS from the IRF7309.

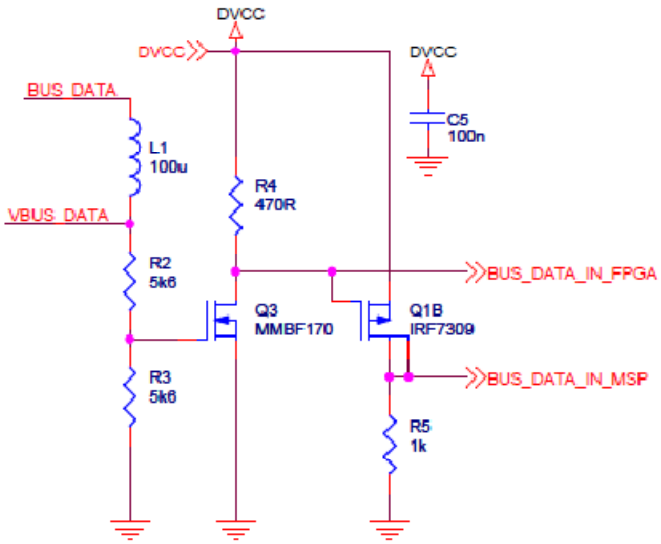

Figure 13 - Bus line decoder (*Slave*).

3.2. Intelligent Peripherals

3.2.1. Overview

An intelligent peripheral consists of the peripheral itself (a sensor or an actuator) connected to a small digital controller (i.e. a microcontroller) and some additional electronics to interface it with the peripheral, an FPGA (in this phase, a development kit) and the digital bus interface. The use of a microcontroller has many advantages and creates a vast amount of possibilities. It has already built in Flash memory, RAM, ADC, DAC, I/O pins, Timers, etc. but it lacks the processing power and the physical BCDFP ALU. So, an FPGA with the same internal structure as the ECU modules is used in conjunction with the MSP to handle the more demanding tasks, leaving the MSP with the sensor interface and diagnostics.

Peripherals that need to be synchronized with the engine's position, such as the ignition coil and the injector for example, have an additional MSP responsible for decoding the synchronization pulses (i.e. Figure 24).

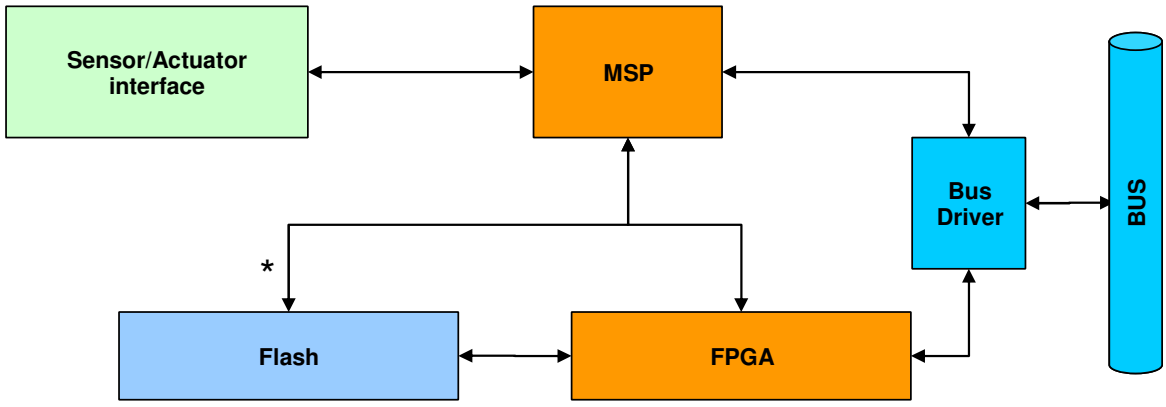

Figure 14 - Peripheral architecture.

3.2.2. Hardware

A PCB board was developed containing the MSP and the Bus Driver blocks from the structure illustrated in Figure 14 in an attempt to reduce problems related to broken wires or bad connections if a prototyping board were to be used.

This board contained the microcontroller MSP430F1611, a JTAG socket to allow connection to the USB programmer, the Digital Bus Driver components, Digital Bus interface with the FPGA board, SPI interface with the FPGA and a prototyping area to design the necessary electronics to interface each specific peripheral.

3.2.2.1. Microcontroller

The chosen microcontroller was the MSP430F1611 from Texas Instruments. Some of the major aspects for choosing this microcontroller were its debugging capabilities and the fast compile/download procedure. The MSP430F1611 is connected to the computer using an MSP USB JTAG programmer.

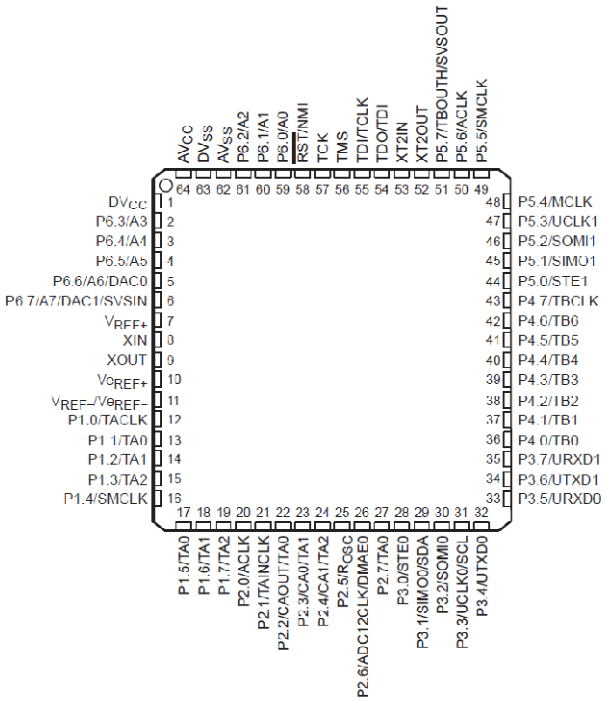

Figure 15 - MSP430F1611 pinout diagram.

Another major aspect was the microcontroller internal characteristics. It is composed by a powerful 16-bit RISC CPU with 125-ns Instruction Cycle Time, 16-bit registers, and constant generators to maximize code efficiency. In the memory field, it has an internal flash memory of 48 KB, 256 Bytes of information memory and 10 KB of RAM. A built-in 16x16 Hardware Multiplier, two 16-bit timers, a fast 12-bit A/D converter, dual 12-bit D/A converter, two universal serial synchronous/asynchronous communication interfaces (USART), I^2C, DMA, JTAG interface, 48 I/O pins and bootstrap capability.

The TQFP package is not ideal for prototyping but a version was available with a breakout board which was more suited for the perforated boards used during the prototype phase. In a later phase, printed circuit boards were developed and the ICs were soldered directly on them.

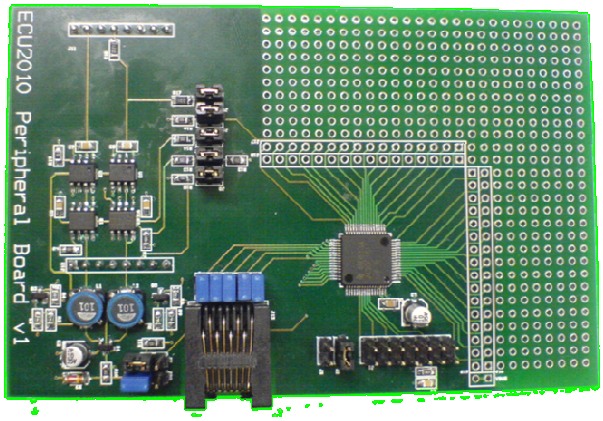

Figure 16 – Peripheral Board.

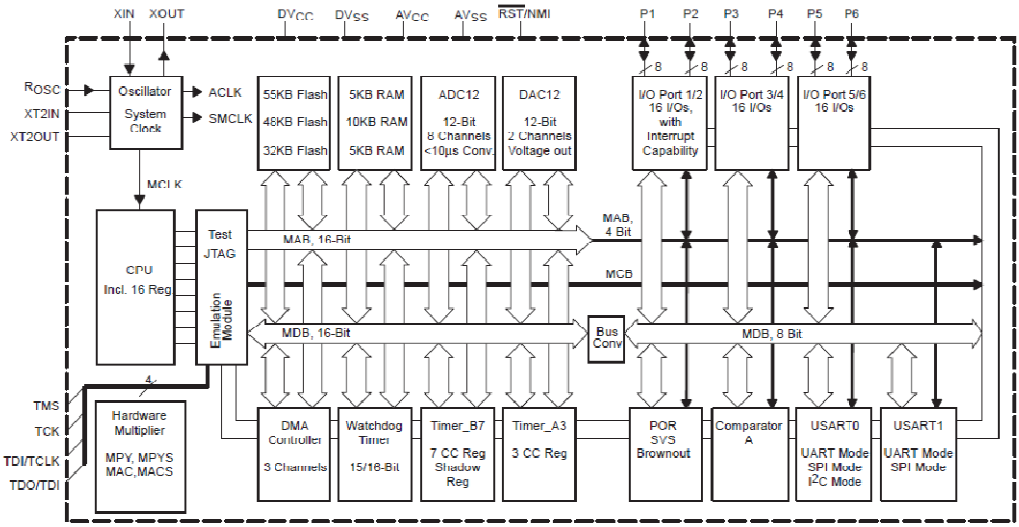

Figure 17 - MSP430F1611 internal architecture.

The microcontroller's main purpose is to act as an interface between the FPGA and the sensor/actuator as this FPGA does not have internal ADC's, DAC's or I/O drivers.

3.2.2.2. FPGA

To maintain the processing topology and power, the peripherals also incorporate an FPGA with the same internal architecture as the ECU modules. The FPGA used is the Spartan 3E XC3S1600E from Xilinx with 1600 logic blocks. For a faster development, development boards were used.

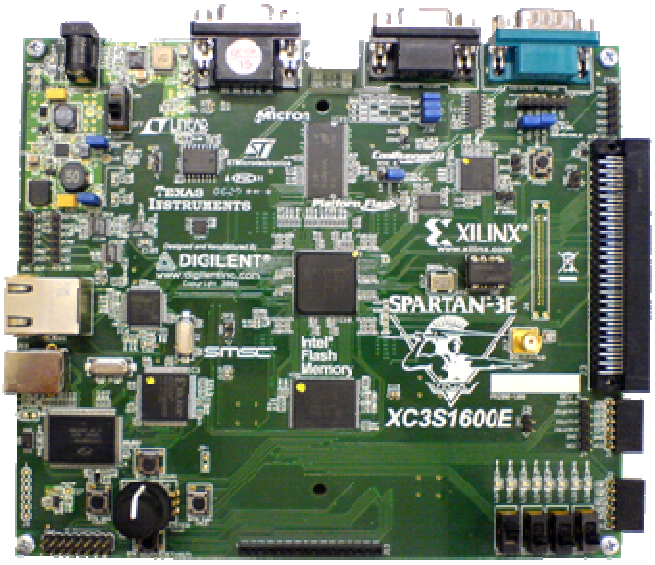

Figure 18 – FPGA kit used. Spartan 3E with 1600 logic gates.

In this document it will be described the Peripheral Manager block. The whole internal architecture is presented in Figure 19 but it is not the scope of this document to describe the remaining composing blocks.

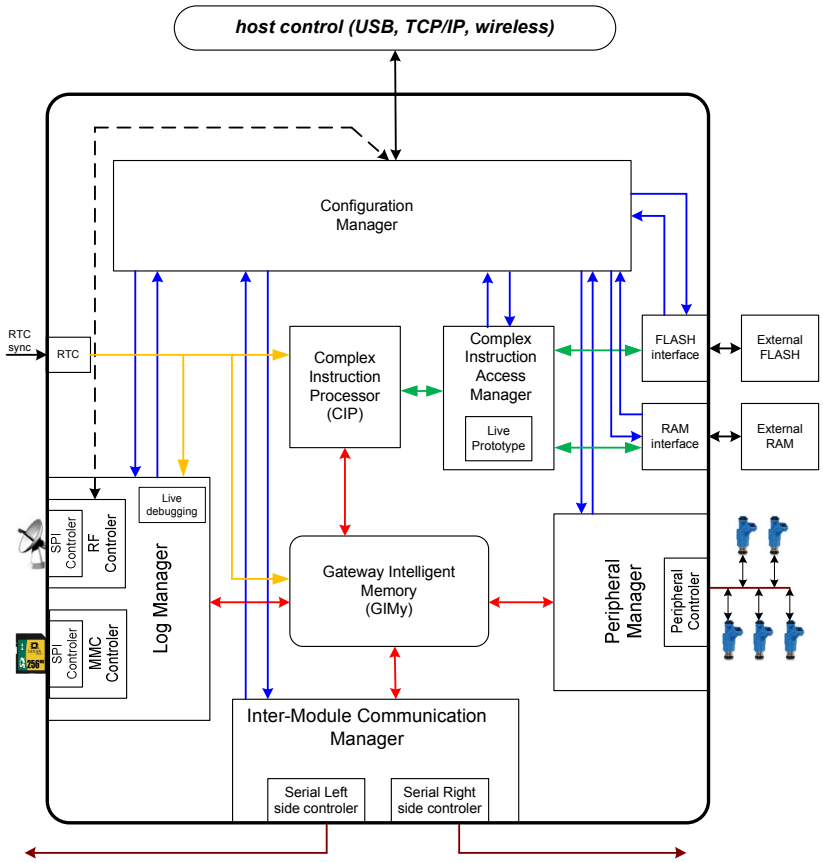

Figure 19 - FPGA internal architecture.

The Peripheral Manager contains the interface between the Peripheral Digital Bus and the internal blocks, GIMy and Configuration Manager. The Peripheral Manager is composed by a block called Peripheral Controller, another called Data Flow Controller and additional interface memories and registers.

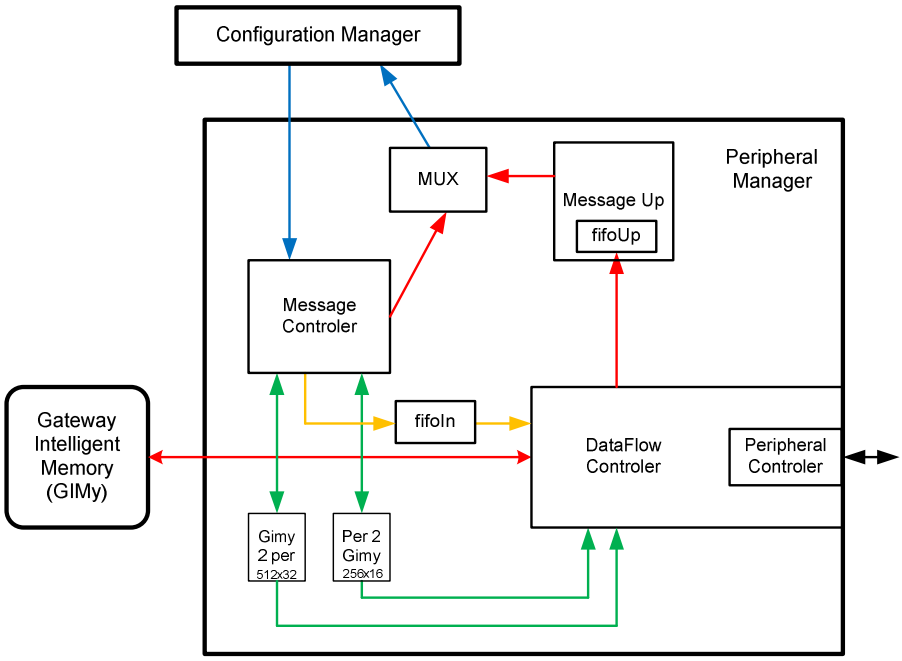

Figure 20 - *Peripheral Manager* block diagram.

Data is fed to and from the Peripheral Controller by the Data Flow Controller. The latter is responsible for scheduling the data requests and data transmissions for the peripherals and routing the incoming messages to the GIMy or the Configuration Manager.

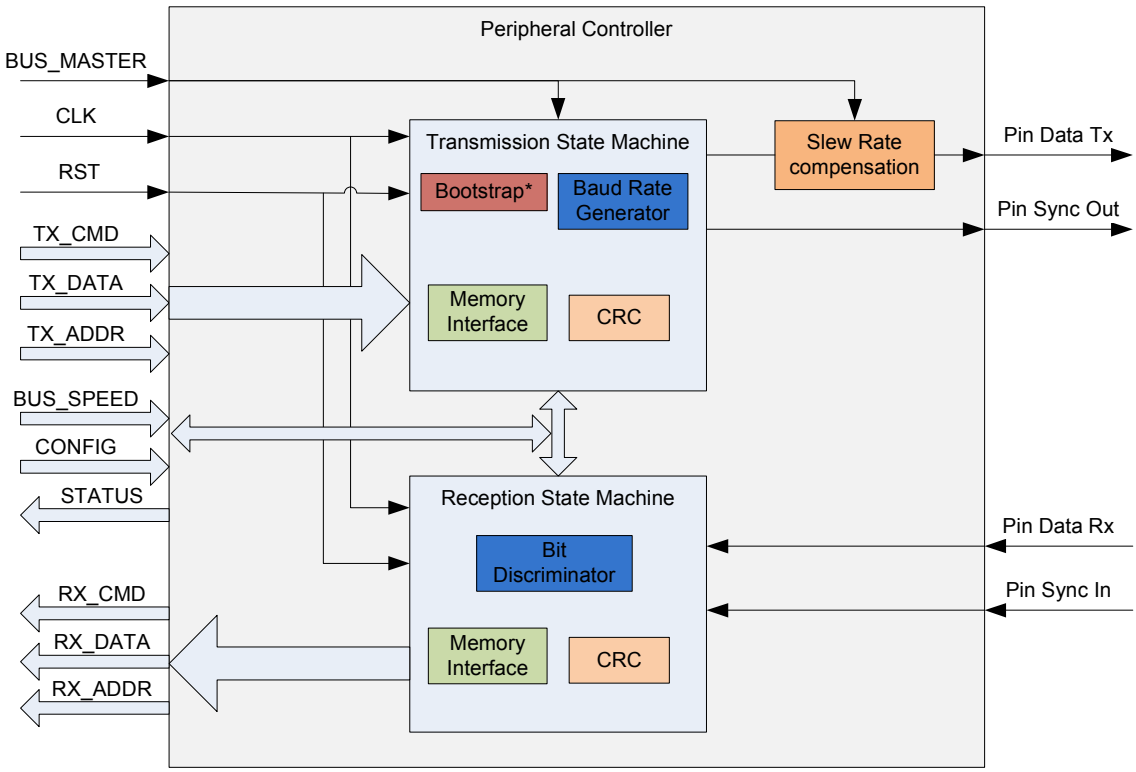

Figure 21 - Peripheral Controller diagram.

This block is responsible for creating the digital bus waveforms. It contains the reception and the transmission state machines which form the stream of bits containing all the fields necessary to form the complete frame, generate the adequate baud-rate and check for errors. The maximum allowed baud-rate depends on the internal clock speed of the FPGA as it is generated by counters that run at that same clock speed. So, the higher the baud-rate for the same clock speed, the lower clock counts per bit are obtained. This means that in the reception block it will be harder to decode the bits, leading to higher error rates. The CRC is calculated in each of these state machines using a parallel architecture. This architecture was chosen because it was easier to implement and to debug than a serial architecture although it uses more logic resources.

The slew rate compensation block is an asymmetrical delay to compensate for uneven voltage rise and fall times. It reads the current desired bus state and when a transition is requested, it applies a delay if necessary and only then applies the transition to the corresponding output pin. This allows more symmetrical waveforms on the receiving device and a lower error rate.

The memory interface blocks contain the logic necessary to read and/or write from the external FIFO memories. These memories act like buffers so that neither part has to wait for the other to exchange data.

The bootstrap capability is used to allow an ECU module to be able to program a peripheral if its internal memory is completely erased. This is achieved through the MSP bootstrap loader. The bootstrap loader allows the MSP to be programmed so that it could then be used to program the FPGA's flash memory.

The bootstrap block generates the specific baud-rate for the bootstrap signals that is about 9600bps and the bootstrap entry sequences. It downloads the firmware to the MSPs internal flash in blocks of 200 bytes.

3.2.3. Sensors

In the engine used to test our system there are several sensor types such as temperature sensors, pressure sensors, throttle position sensors, crankshaft position sensors and push-buttons. Some have analog interface like the temperature and the pressure sensors, and others have digital interface as the crankshaft position sensor or the push-buttons. Each one has specific electric circuit interface to the microcontroller depending on its interface.

3.2.3.1. Absolute Angular Magnetic Encoder

To control the combustion process, the ECU needs to know, with certain accuracy, the engine's crankshaft position. The fuel has to be injected in a precise quantity and has to finish in a pre-determined angle so that the air-fuel mixture can be correctly formed when the spark is fired. The spark firing angle has even tighter angle requirements. The spark coil has to charge up to the desired current and has to fire the spark in the exact crankshaft angle defined by the ECU.

Figure 22 - Inductive sensor (Left)[9], Digital Hall sensor (Center)[10] and Flywheel (Right)[8].

The actual method used to acquire the engine's position is based on a flywheel (as in Figure 22) with several teeth (usually 60), equally spaced between them, and an inductive or a Hall sensor to detect the passing teeth. The inductive sensor is a passive sensor while the Hall sensor is an active sensor and outputs logical values. To determine

the Top Dead Centre (TDC) (position of the crankshaft where the piston is at the top of the cylinder), usually one or two teeth are omitted from the wheel. In some applications, where more precision is needed, i.e. development benches, optical encoders are used which can achieve resolutions of one degree, instead of the 6 degrees provided by the flywheel approach. The optical encoder has another positive point. The ECU can at any time, and even with the engine stopped, know its position whereas with the flywheel the engine must be rotating so that the ECU can detect the TDC and synchronize itself with the crankshaft.

These methods have both a common disadvantage. The 4-stoke combustion cycle requires 2 crankshaft rotations (720 degrees) to complete and they cannot distinguish from which one of them the engine is in a given moment. So, the ECU needs to have an extra sensor in the camshaft to allow it to know in which of the 360 degrees cycle the engine is. Also, the optical encoder has better resolution than the flywheel but lacks the physical robustness which keeps it from being massively used.

A viable alternative could be an Absolute Angular Magnetic Encoder. The specific sensor used was the AM8192B from RLS[11].

Figure 23 - *AM8192B IC* and final product aspect from RLS.

This sensor allows contactless angular position encoding over 360 degrees with 13 bit resolution and a maximum rotational speed of 60.000RPM. It also has the advantage of providing the current angular position when the engine is stopped which means that the ECU knows exactly which cylinder will be ready for explosion in a start-up condition. Having 13 bits of maximum resolution it can be directly connected to the camshaft instead of the crankshaft. This reduces the available resolution to a half but enables the ECU to have position information over the complete combustion cycle making a secondary sensor unnecessary.

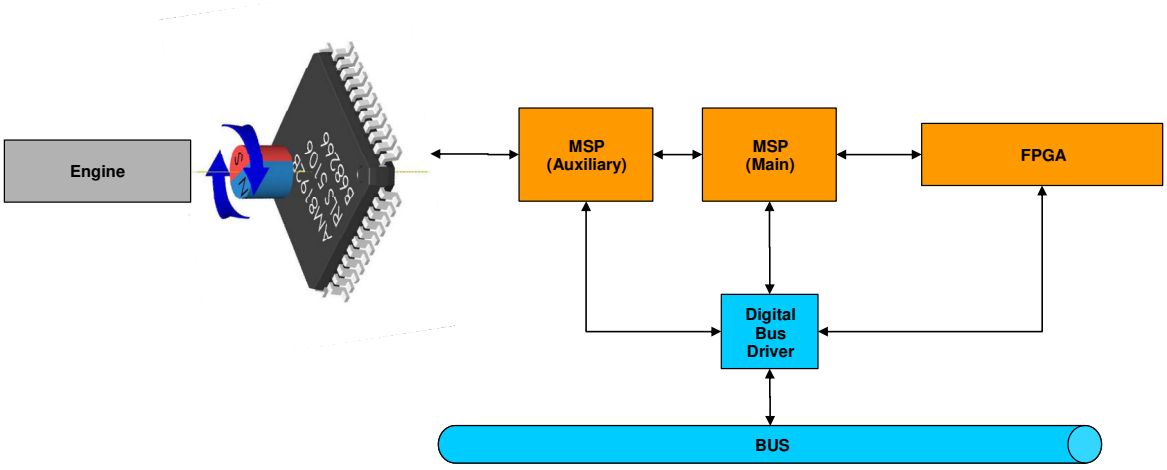

Figure 24 - Block diagram of the crankshaft sensor.

The sensor has an absolute binary synchro-serial interface (SSI) which allows the reading of the actual magnet position and an incremental output. Using the serial interface, the current position data (up to 13 bits) can be read at a maximum rate of about 4Mbit (4MHz clock).

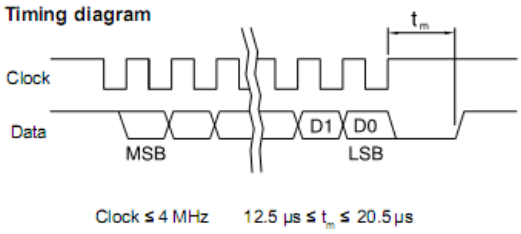

Figure 25 - Timing diagram of the *SSI* interface.

Although the SSI interface allows the current magnet angle to be known, it has the inherent delay of the digital communication bus. This delay is not desirable because it will cause the engine to not perform at the top performance, have faulty running or even be destroyed. To bypass this issue, the sensor's quadrature interface is used. It has 3 outputs, A, B and Z (or R_i). Outputs A and B are two digital signals shifted by 90 degrees from one another as shown in Figure 26. The Z output is the reference signal. It goes 'high' when the magnet is in the zero angle position.

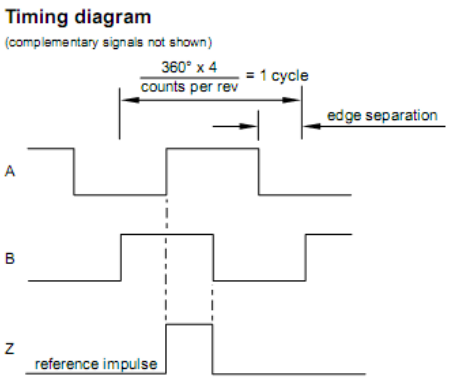

Figure 26 - Timing diagram of the quadrature outputs.

There is still the problem of sending the information of these three outputs using a single wire. To convert these waveforms into a single waveform capable of being sent to the ECU and all other peripherals that need this information through the SYNC line of the peripheral bus, an auxiliary microcontroller was introduced in the Smart Peripheral. This MSP detects each of the transitions of A and B outputs and converts them into formatted pulses. Each pulse represents a sensor count and a pulse with the double of the time represents the zero angle position (Figure 27). As the MSP only supports either rising edge or falling edge (not both simultaneous) interrupt generation on a single pin, each one of the quadrature signals (A or B) is connected to two interrupt enabled pins (Port 2), being one of them configured to generate an interrupt in the rising edge and the other in the falling one.

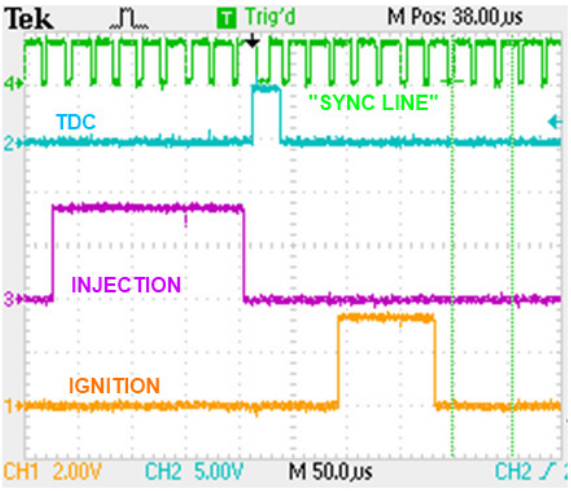

Figure 27 – Synchronization line waveform. Injection and ignition tests.

The synchronization of the entire combustion cycle is solved using output Z. When the MSP detects a pulse of the Z output it generates a longer pulse on the bus. This allows all other devices to acknowledge that the combustion cycle has ended and that a new one is beginning.

The two microcontrollers are connected to each other through a SPI bus running at 1 MHz. This bus is used to exchange configuration parameters including the current crankshaft position.

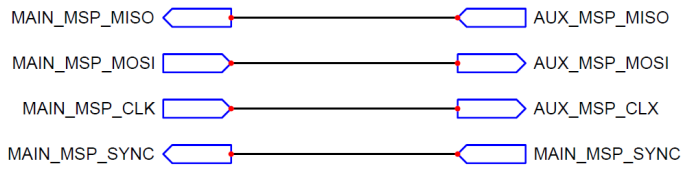

Figure 28 - *SPI* connection between main and auxiliary *MSP's*.

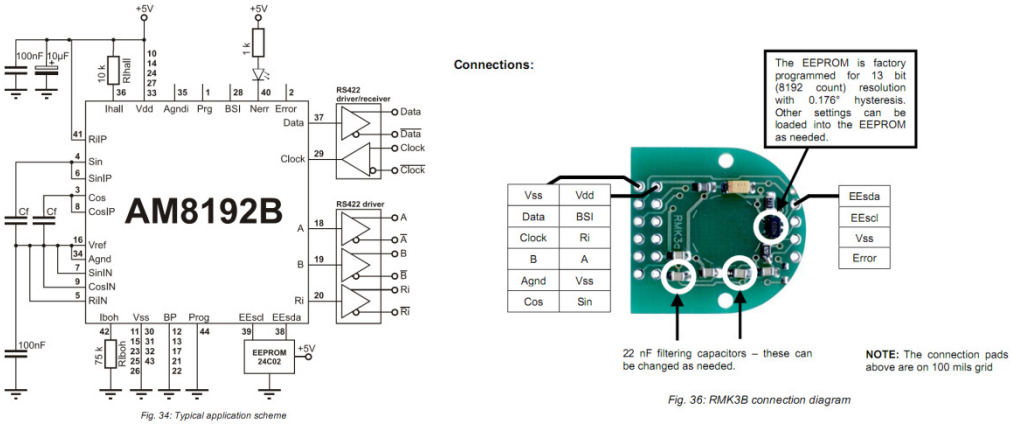

Figure 29 – Typical application scheme (Left) and sensor kit and pinout (Right).

3.2.3.2. Temperature sensor

On the engine setup there are two temperature sensors, the air temperature and the oil temperature sensors. Both are resistive Negative Temperature Coefficient (*NTC*) type which means that the resistance across its terminals decreases as the temperature rises. Also, the relation between them is not linear, which means that it has to be linearized in the software using a more complex formula or a lookup table. Air temperature precision should be up to 2 degrees while the oil temperature is less critical and can have up to 10 degrees.

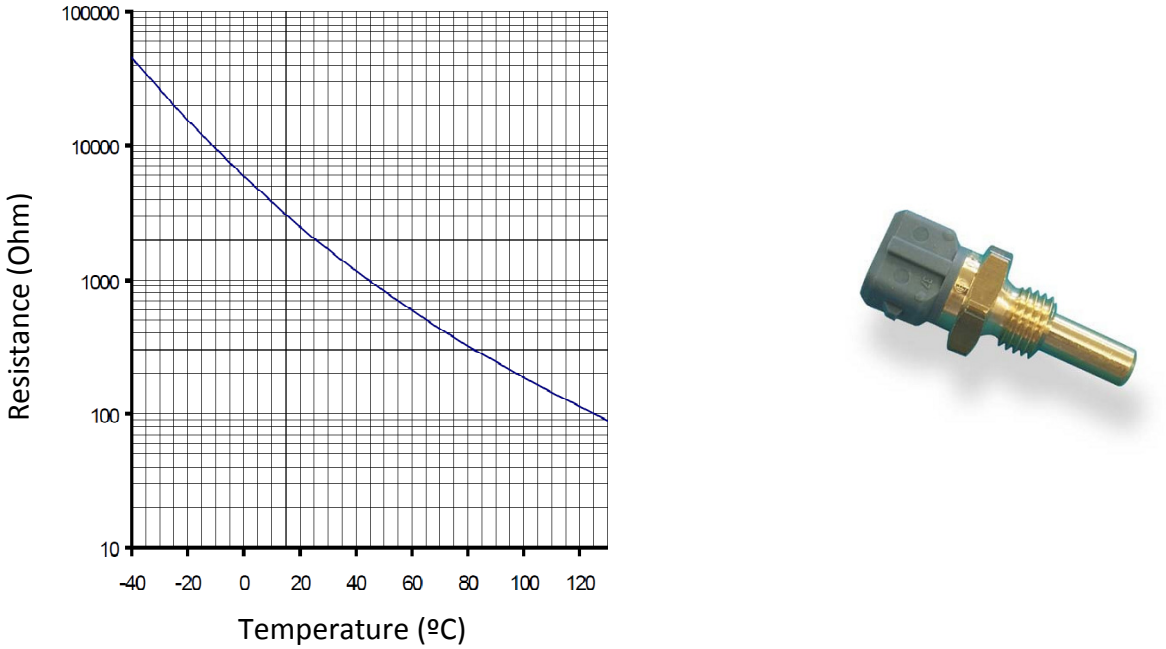

Figure 30 - Linearity curve (Left) and Temperature sensor (Right) [16].

The first method offers better resolution but uses much more processing power and is slower. So the latter method was used that, although uses a more memory, is faster.

The sensor should be polarized using a constant current source but this would raise the production costs, so a simple pull-up resistor was used, keeping in mind the compromise between low current and good resolution.

Temperature in °C	Resistance in Ω			Temp. in °C	Resistance in Ω	
	min.	nom.	max.		min.	max.
-40	40901	45313	49725			
-30	23817	26114	28411			
-20	14236	15462	16688			
-10	8727	9397	10067	-10 ± 1	8325	10572
0	5520	5896	6272			
10	3576	3792	4008			
20	2375	2500	2625	20 ± 1	2280	2736
30	1610	1707	1803			
40	1102	1175	1247			
50	777.7	834.0	890.2			
60	552.4	595.5	638.6			
70	402.1	435.7	469.3			
80	296.3	322.5	348.8	80 ± 1	287.7	359.1
90	222.4	243.2	264.0			
100	169.9	186.6	203.4			
110	130.7	144.2	157.6			
120	101.8	112.7	123.6			
130	80.35	89.3	98.25			

Figure 31 - Temperature to resistance table.

3.2.3.3. Intake air pressure sensor

The intake air pressure sensor was used to measure the air pressure on the intake manifold. The intake pressure is used by the *ECU* to measure how much power the engine is producing. Lower manifold pressure means more air entering the combustion chamber, more fuel delivered by the *ECU* and therefore, more power. Its interface is analog, but it's an active sensor. The used sensor has a 3 connections, power (+5Volts), analog output and ground.

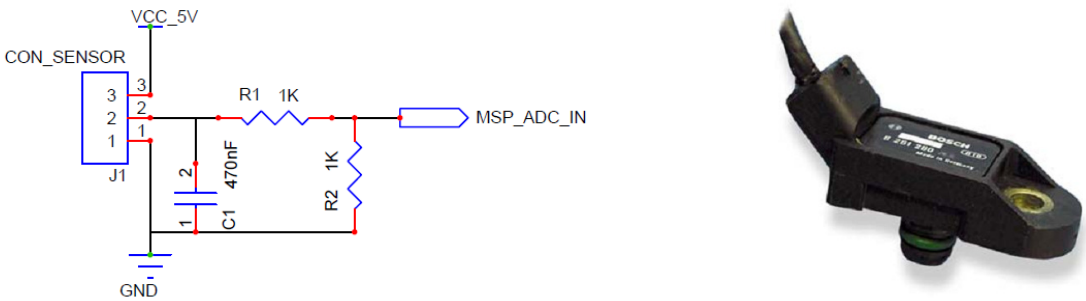

Figure 32 - Sensor interface schematic (Left) and Intake Air Pressure sensor (Right) [17].

3.2.3.4. Fuel pressure sensor

To calculate the exact time that the injector needs to be open so that the desired amount of fuel enters the combustion chamber, the *ECU* needs to know the pressure in the fuel line. This task is performed by the fuel pressure sensor. The sensor used is the 'Pressure Sensor Fluid PSS-10' from Bosch and is able to measure fluid pressures from 0.5 to 11bar (absolute pressure).

Figure 33 - Fuel Pressure sensor.

The sensor is supplied with 5 Volts and has a linear response with a slope of 400mV/bar. The output has an offset of 100mVolts and is connected through a resistive divider to the microcontroller's *ADC* input.

3.2.3.5. Throttle position sensor

In the setup used, the throttle was controlled externally to the *ECU*. This control was made by a servo motor attached to the intake valve and controlled by an auxiliary board. Also attached to the intake valve is the throttle position sensor. It's a 'Rotary Potentiometer RP 308' with 5kOhm total resistance and a measuring range of 308 degree. It was powered with 3.3 Volts so its output could be connected directly to the microcontroller's *ADC*. Capacitors were added to filter external noise generated mainly by the engine.

Figure 34 – Throttle position sensor

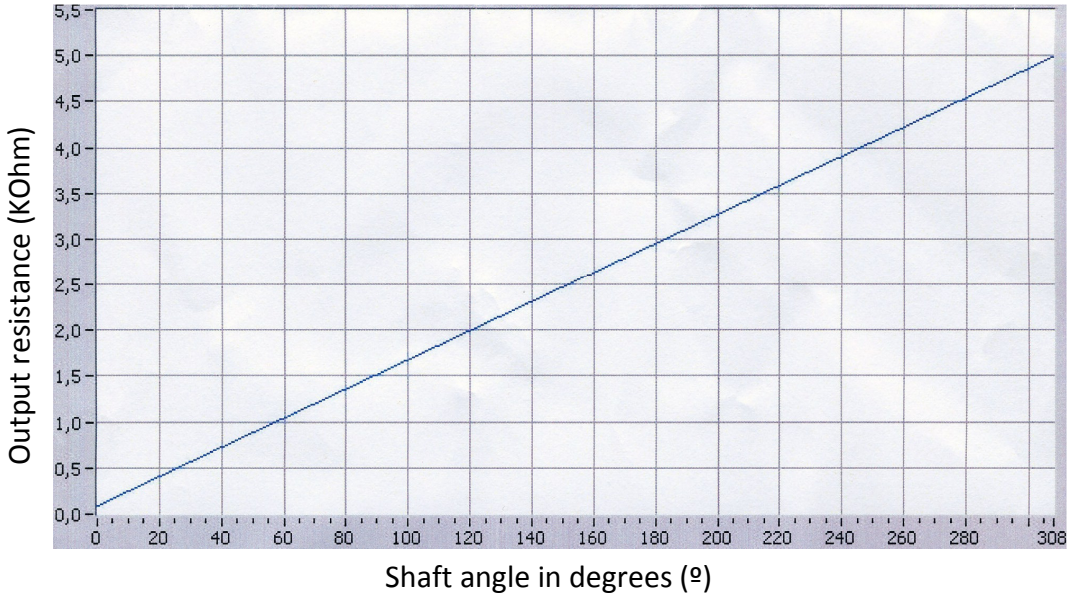

Figure 35 – Sensor linearity curve.

3.2.3.6. Battery Voltage sensor

This sensor is responsible for reading the engine's battery voltage. This reading is important to calculate the amount of the time that the ignition coil and the fuel injector need to be active to achieve the desired output values. This is important during start-up because the starter draws a lot of current from the battery, reducing its voltage. If the battery has low charge this voltage can be as low as 6 volts for a 12 volt battery so the *ECU* has to take this in account when controlling the injection and the ignition. As the control unit has a dedicated power supply, the ground connections need to be separate so that there is no noise from the engine affecting the *ECU*.

The battery voltage line has a lot of noise and voltage spikes resulting from ignition coil commutations and other actuators. Regarding this issue, it's a good practice to isolate the battery line from the more sensitive electronics in the sensor. Digital isolators are not applicable here as the unit to measure is analogue. In this case an isolation amplifier is recommended. The *HCPL-7800* Isolation Amplifier was used. It is somewhat similar to a normal Operational Amplifier but has its output isolated from the input. It enables the measurement of the battery voltage with the *MSP's* internal *ADC* without the risk of damaging it. At the output of the *HCPL-7800*, an LM358 Operational Amplifier is responsible for the conversion from differential to single-ended signal.

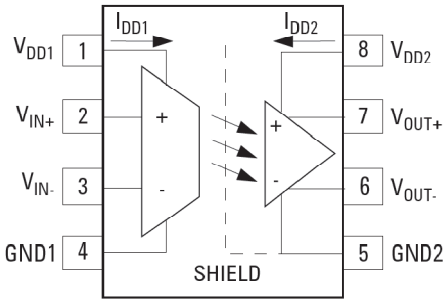

Figure 36 - *HCPL-7800* Isolation amplifier.

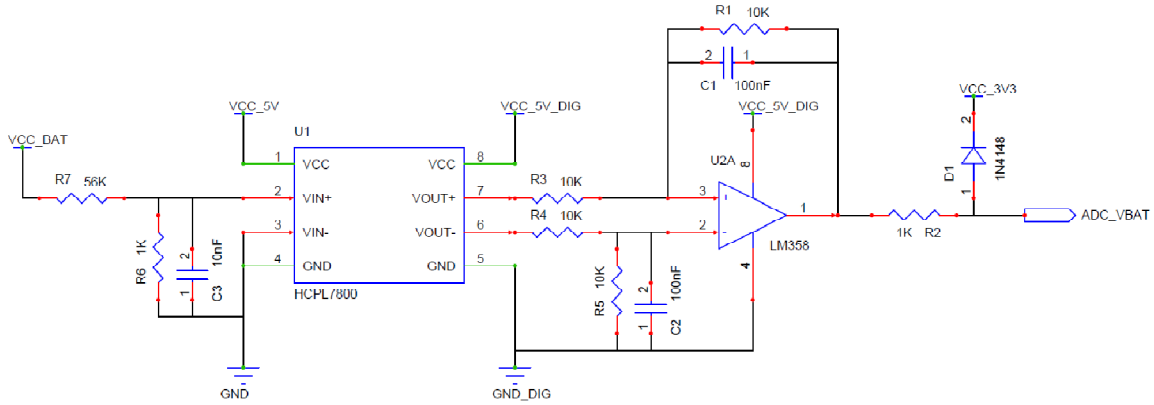

Figure 37 - Implemented isolation circuit schematic.

3.2.3.7. Control interface

These are the switches used to control the engine supply and start-up. They are connected to digital pins on the microcontroller with an additional pull-up resistor. The circuit is simple and is described in Figure 38. The ignition switch is a single pole single throw switch that stays on when it is desired to run the engine. The starter switch is a single push button that is pressed to power the starter motor and turn the engine.

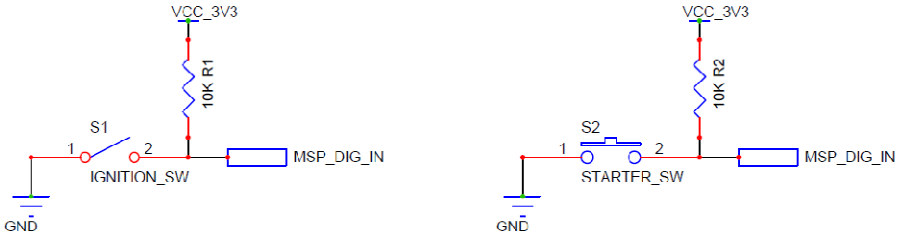

Figure 38 - Control interface.

3.2.3.8. Lambda Sensor

A Lambda sensor was used during system setup to find the correct injection and ignition table values. This sensor makes it possible to know if the air/fuel mixture is being made in the right proportions (14:1) inside the combustion chamber by analysing the exhaust gases. It was solemnly used for calibration purposes and it was not included in the ECU functions, so it was not further developed.

3.2.4. Actuators

To control the combustion process the *ECU* needs to act in the engine's parameters.

The main actuators used to control the engine are the air intake valve, the ignition coil and the fuel injector. The air intake valve controls the air mass that enters the cylinder chamber, allowing more fuel to be delivered and increasing the engine's output.

3.2.4.1. Ignition Coil

In a Gasoline engine, for the air-fuel mixture to ignite, there has to be an ignition source. This ignition is a spark generated by the spark plug that receives a very high voltage pulse generated by the ignition coil. The main variables involved in the generation of the spark are the ignition angle which is the angle where the spark has to take place, and the ignition coil charge time that defines how much energy the spark will have. The ignition angle is a critical variable. If the spark is fired too late, the engine doesn't produce the full power, if it's fired too soon the cylinder can try to turn the crankshaft backwards and destroy the engine.

Figure 39 - 'Single Fire Coil PS-T' from Bosch [14].

The Ignition Coil used was the 'Single Fire Coil PS-T' also from *Bosch* that had an integrated power stage which simplifies the interface, only needing a digital signal with low current drive to turn it on (10 to 20mA). It can generate a spark with a maximum energy of 42miliJoule and a maximum voltage of 27kV.

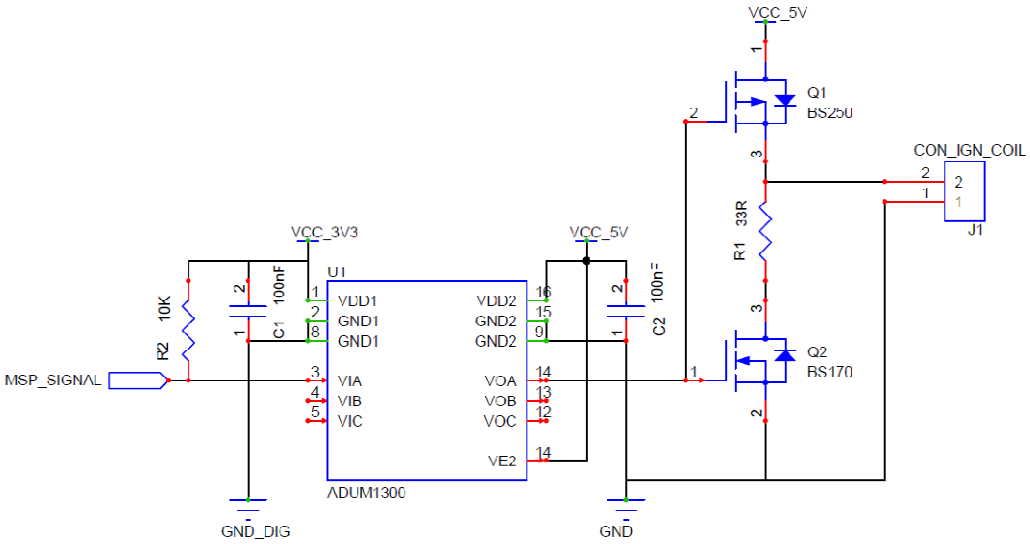

Figure 40 - Isolation and drive of the control line for the Ignition Coil.

3.2.4.2. Injector

The injector has the function of controlling the fuel flow into the cylinder chamber and atomizing the fuel into small particles that allow a better fuel-air mixture. It acts like an electrically controlled fuel valve. There are mainly two types of injectors, the solenoid and the piezo. The main parts in an injector are the pressurized fuel input, the solenoid or the piezo element, the plunger, the output nozzle and the chassis. In the first type the fuel flow is controlled by a solenoid that pulls the plunger which is blocking the output nozzle. When the solenoid is not powered, a spring pushed the plunger against the output hole and closes it, stopping the fuel from flowing to the chamber. On the piezo type, there is no need for the spring because the piezo material expands and contracts when there's voltage applied or not, respectively.

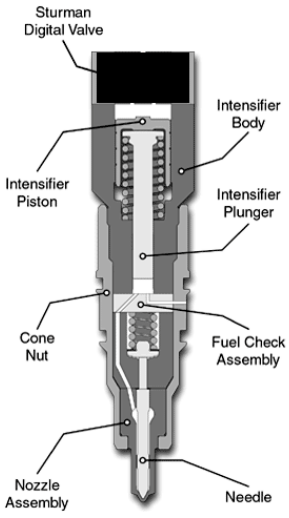

Figure 41 - Injector internal components[12].

The injector was supplied by the battery voltage and

driven by the power *MOSFET BUZ11*. The switching command signal comes from the auxiliary *MSP* and is fed to the *ADuM1300*. The *ADuM1300* is a general purpose three channel digital isolator from Analog Devices. It uses a proprietary technology based on high speed *CMOS* and monolithic transformers which provides better performance and consumption than optical based isolators. Q1, Q2 and Q3 provide level shift and the necessary current to completely turn Q4 on. Q3 was later added because of a problem that only became visible during implementation. If the power supply of the *ADuM1300* from the microprocessor side (*VDD1* in this case) is turned off with *VDD2* still active, the outputs will go high, turning on the injector. Q3 acts as an inverter and prevents the latter.

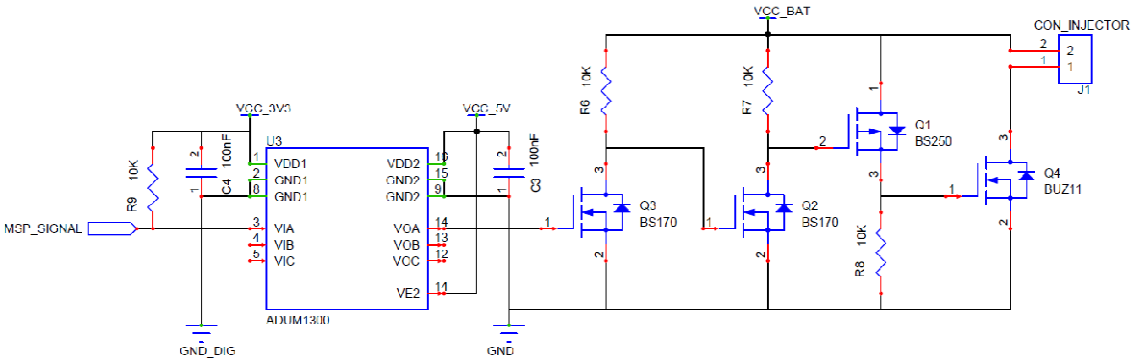

Figure 42 - Isolation and high power drive of the Injector.

3.2.4.3. Fuel Pump

The Fuel Pump is responsible for delivering fuel from the tank (normally on the back of the vehicle) to the injection circuit in the engine. It has to be turned on before the engine is started so that the fuel pressure can build up and be in the nominal value when the engine starter is turned on. It stays on for a few seconds when the engine is turned off so that it can start back again quickly in case of a stall, for example. The activation circuit is simple, normally being activated by a relay that is controlled by the *ECU*.

'Fuel Pump FP 100' from *Bosch* was used. It has a fuel delivery rate of more than 100l/h and generates an output pressure of 5bar.

Figure 43 - 'Fuel Pump FP 100' from *Bosch* [13].

The nominal working voltage is 13.5V and the current consumption reaches 5A at 5bar output pressure. The control signal comes from a circuit similar to the one used to drive the injector, only this time the relay is the load.

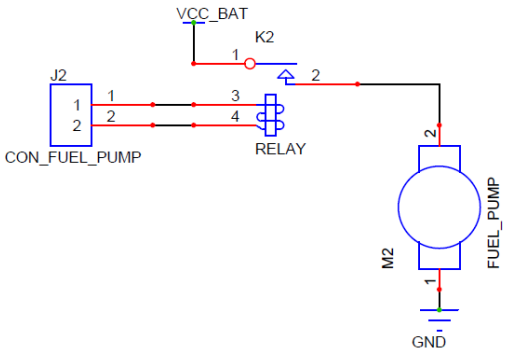

Figure 44 - Fuel pump interface schematic.

3.2.4.4. Engine Starter

For the engine to run, it first has to be set in to motion. This is accomplished by an electric motor with low *RPM* but very high torque. This motor couples itself with the engine's crankshaft when it starts to rotate and decouples when it stops as it is only intended to start the engine. To achieve so much torque, it consumes a lot of current reducing the battery charge quickly. So, it should only be on during the minimum time necessary for the engine to start.

The connection schematic is identical to the fuel pump. It has the same *MSP* interface, with the *ADuM1300* and the *MOSFETs*, and a relay similar to the fuel pump but with higher current capability.

Figure 45 - Engine starter [15].

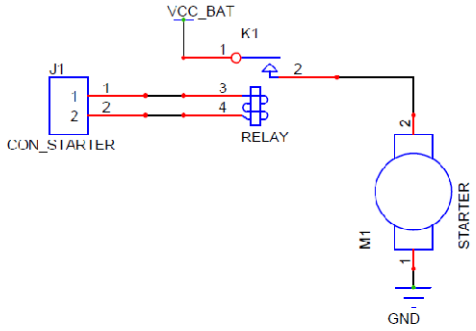

Figure 46 - Engine starter interface schematic.

3.3. Software

3.3.1. Overview

The software code used in the microcontrollers and the peripheral part of the Integrated Development and Management System (IDMS) will be explained here. Also, during the development of this project, some software tools were used and they will be mentioned as well.

3.3.2. Peripheral Firmware

The peripherals firmware code was written in C language using *IAR Embedded Workbench* tool for MSP430 [18]. The IAR IDE has an integrated code editor, compiler, programmer and debugger that allow the user to have all the capabilities it needs in just one application while maintaining a stable and reliable operation. This fact helped to reduce the overall development time.

The firmware was written to be virtually identical for all peripherals because if a problem occurred in one of them so that its firmware needed to be downloaded again, it wouldn't be necessary to remove it from the bus and from its location to be re-programmed. This is due to the fact that during firmware update there was no way of distinguishing one peripheral from the others on the same bus. Different type peripherals required different code and this is accomplished using a global variable that enables/disables certain functions or parts of the code.

The software structure is illustrated in Figure 47. This implements the basic functionality of each peripheral such as internal temperature reading, internal voltage monitoring, lifetime counter and electric diagnostics. The internal diagnostic variables are stored in the MSP's internal Flash memory. As this memory has limited read/write cycles, the variables are only written each 10 minutes, allowing longer memory life while maintaining a certain precision in the case of the lifetime counter. The limited read/write cycle count issue can be solved using an external EEPROM memory or a microcontroller that has an internal EEPROM memory.

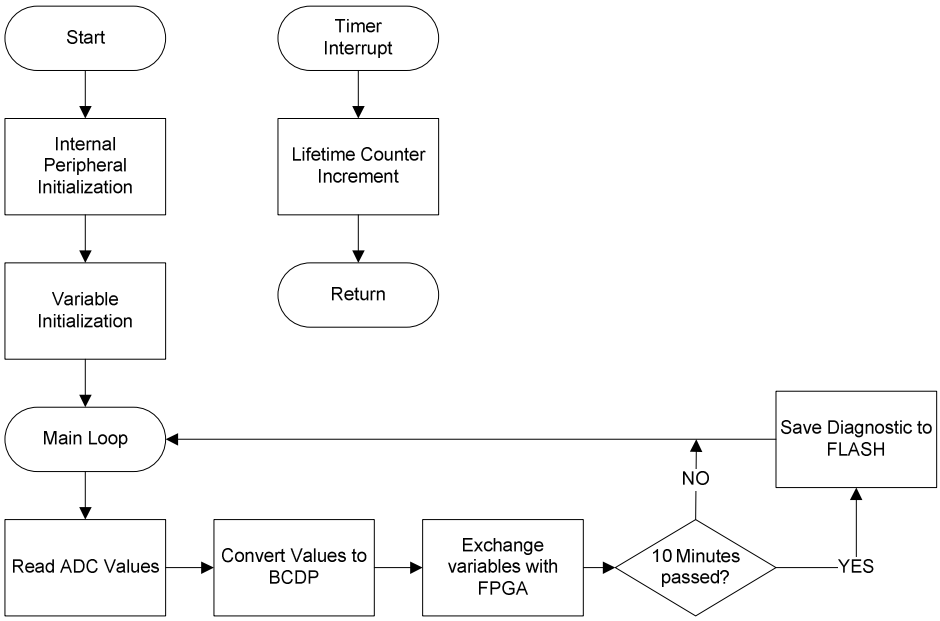

Figure 47 - MSP software diagram.

In the case of the auxiliary *MSP's*, the software has a different structure as they do not need most of the functionalities which are already implemented in the main *MSP*.

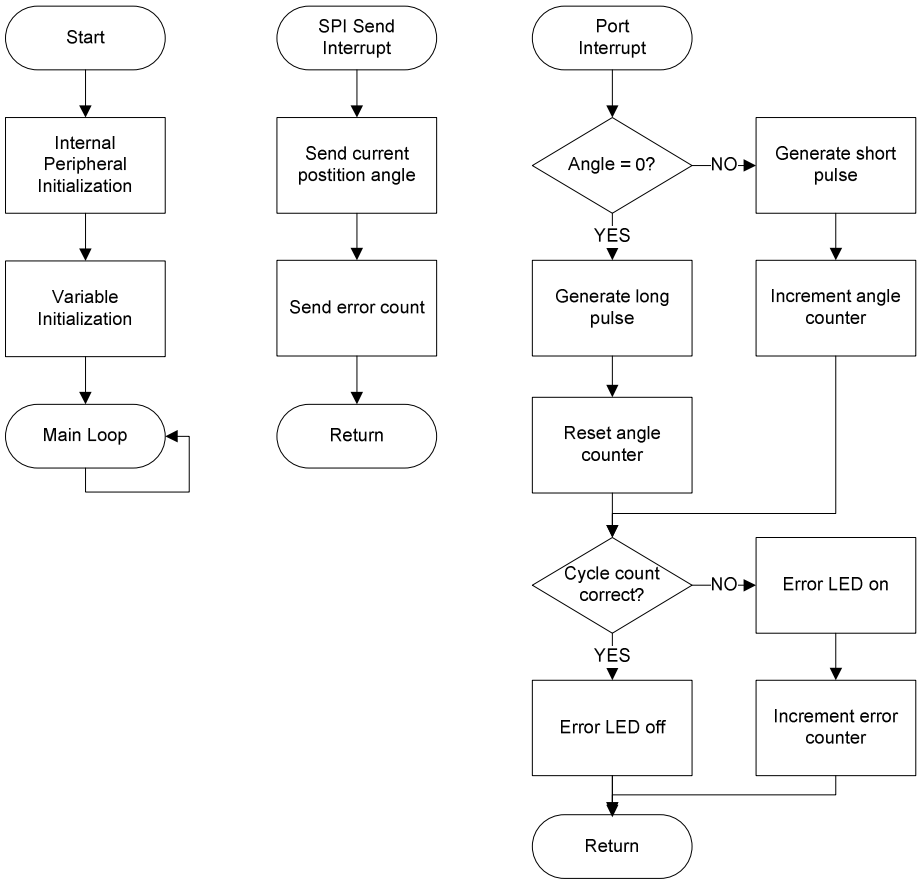

Figure 48 - Software diagram of the Magnetic Encoder auxiliary MSP.

In this case, the main loop has no code. All of the tasks are executed in the respective interrupt routines.

Angle counting is made though a local variable that is incremented at each transition of the sensor's quadrature output (A or B). These transitions generate Port2 interrupts. At each interrupt a short pulse is generated in the *SYNC* line of the digital bus. When the zero angle pulse (Z sensor output) is detected the counter is reset and a long pulse is generated instead, signalizing the peripherals that an engine cycle has been completed. This pulse allows the peripherals to synchronize their internal angle counters. If the number of pulses in a cycle is not equal to the predefined value, an error counter is incremented and a visual indication is produced (LED).

In the *SPI* interrupt the transmission frame is formed. The frame is composed of 6 bytes where 4 of them are data and 2 are frame validation bytes. This produces a significant overhead but this is not critical in this bus. Validation bytes are composed by a 0xAA in the beginning of the frame and a 0x55 in the end. The other 4 bytes correspond to 2 values (2 bytes each), one being the current position angle and the other the cycle error count. This enables the main *MSP* to know if the angle transitions are being correctly detected.

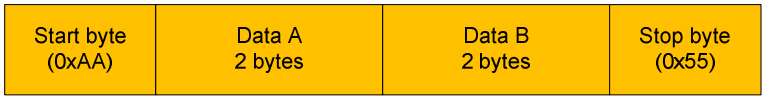

Figure 49 – SPI data frame.

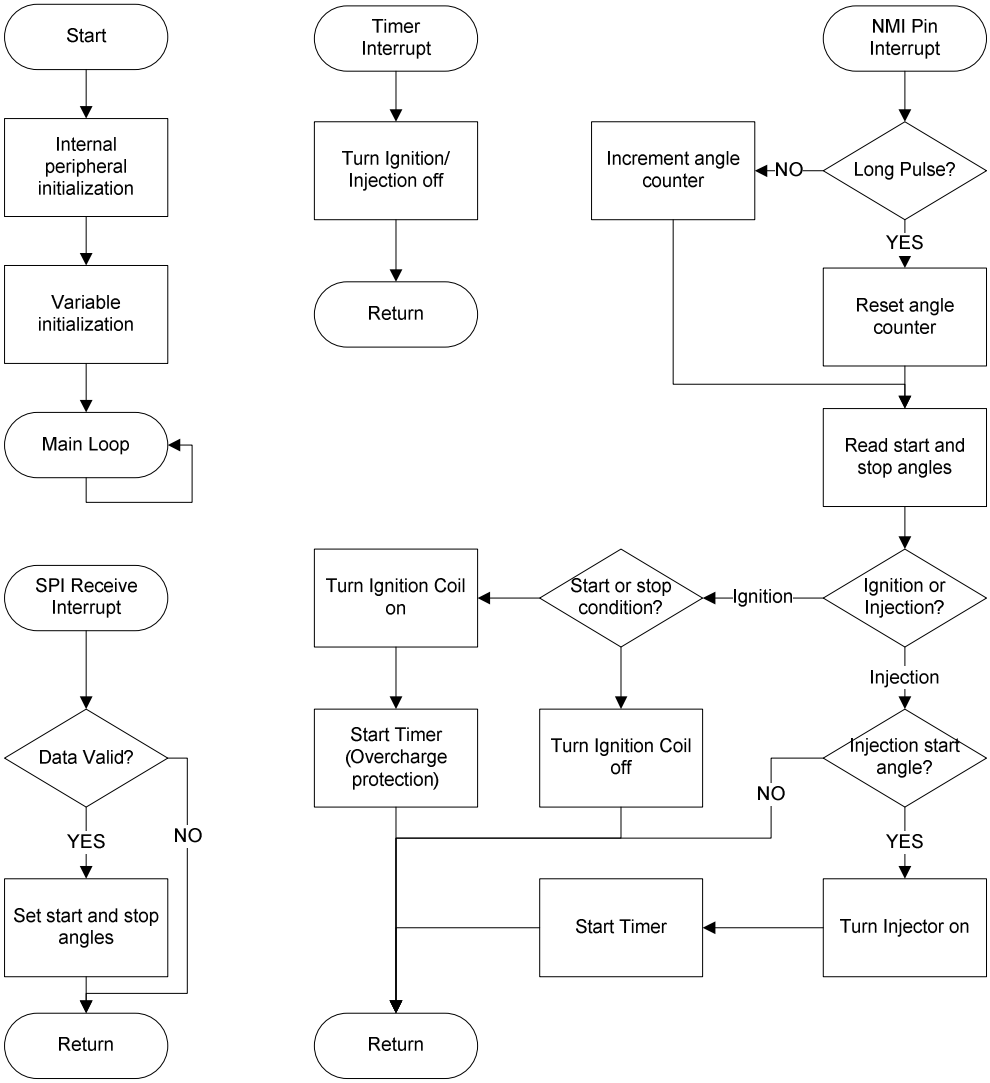

Figure 50 - Software diagram for the auxiliary *MSP* of either the Fuel Injector or the Ignition Coil.

Again, in this auxiliary *MSP* the main loop doesn't execute any function. Here, the *SPI* interrupt is a reception interrupt. The data bytes are 2 integers that represent the start and stop angles (or injection time). Frame validation bytes are checked and data is only read if they are correct. To avoid updating the two variables while they are being used, causing possible data inconsistency, they are copied to temporary variables and are

only updated right before being used. As the firmware for either Ignition or Injection auxiliary *MSPs*, they are distinguished by an input pin that is either connected to *Vcc* or to *Ground* respectively. If it's the Injection auxiliary *MSP* and the current angle is the start angle, the injector is turned on and the timer is loaded with the desired injection time and started. Then, in the timer interrupt routine, the injector is turned off and the injection cycle is complete.

If the pin reading corresponds to an ignition auxiliary *MSP*, and also, the start angle is reached, the ignition coil is powered and the timer is loaded with a pre-defined delay (about 10miliseconds). This time the timer acts like a safety measure, preventing the ignition coil from being powered for too long leading it to overheat and ultimately to its destruction. When the stop angle is reached, the ignition coil is powered off and the ignition spark is produced in the cylinder chamber.

3.3.3. Integrated Development and Management System (IDMS)

The Integrated Development and Management System (*IDMS*) is the software tool that allows the user to control and monitor all the processes involved with the ECU. It integrates several tools such as function graphical design, graphical peripheral location, global system view, peripheral diagnostic information, etc.

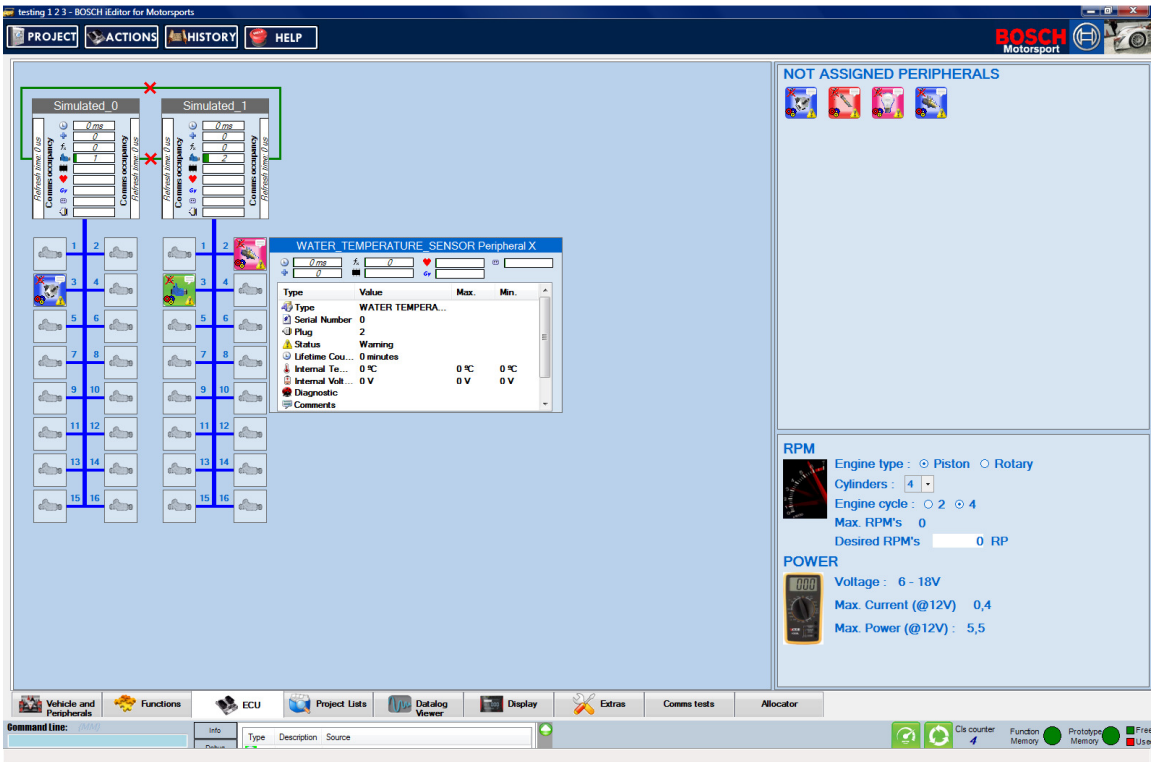

Figure 51 - IDMS – ECU tab view.

In the '*ECU*' tab, the user has access to the whole *ECU* architecture. Here there are three main areas, the *ECU* topology view, the set of not assigned peripherals and the information about the speed attained by the system and its power consumption. In the topology area it is shown how the modules are connected inside the *ECU* and the corresponding peripherals connected to each one of them. In each module there are graphic bars that indicate their internal state. They show the number of functions and macros allocated to each one, the total cycle time, the number of peripherals connected and the communications speed. It also shows the occupancy of the internal memory (*GIMy*), the Live-Prototype memory and the log memory.

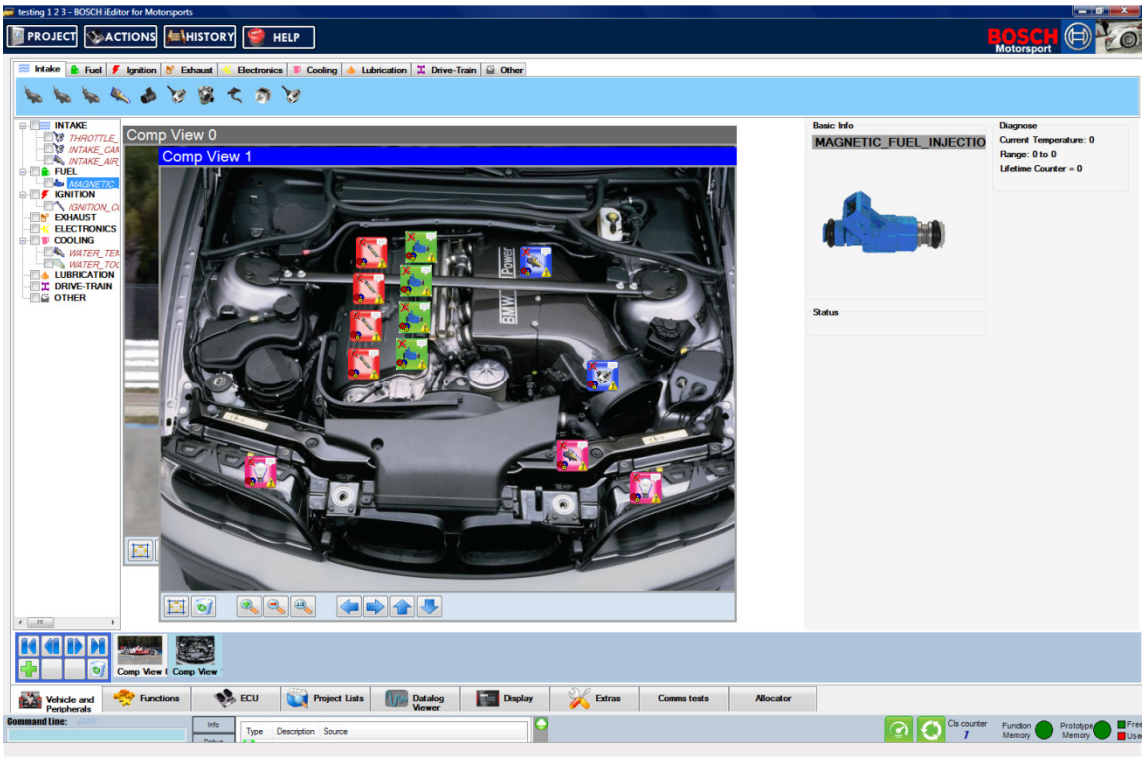

Figure 52 - *"Vehicle and Peripherals"* view.

The peripherals are represented by small icons. These icons include an image that illustrates a generic peripheral of that type, the background colour representing the group at which the peripheral belongs and four mini icons that show the internal diagnostic state. The upper left icon signals whether the peripheral is already used in a function or not. The upper right icon indicates if the peripheral has any user assigned comments. On the bottom left icon it is shown whether the peripheral is being simulated or if it's a real peripheral and if it is plugged in or not. Finally, the right bottom icon shows the status of the peripheral. If this icon is a green symbol, everything is working properly, if it shows any other coloured icon means that something is not working properly.

Whether the wrong peripheral is connected in a specific bus or plug, the serial number has changed, if there are any important diagnostic warnings or other issues, will change this icon into a warning sign.

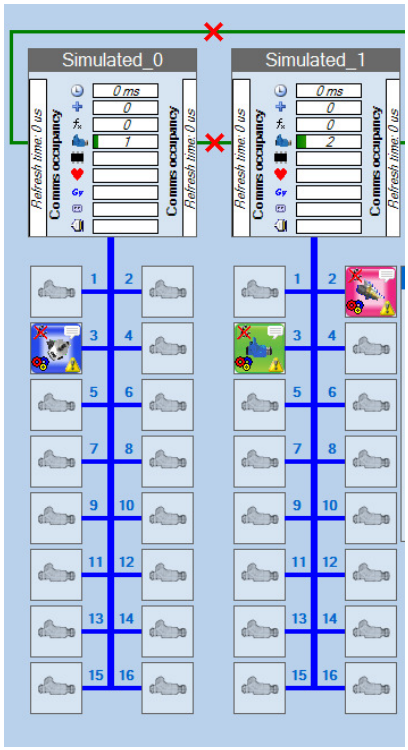

Figure 53 – ECU topology.

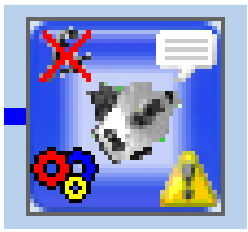

Figure 54 - Peripheral icon.

More detailed information about the peripheral is available by hovering or left-clicking on the icon. This shows a small window containing graphic bars that show the internal information similar to the ones on the module, like cycle time, number of functions and macros, etc. and a text box that contains peripheral information and internal diagnostic, for example, the serial number, the lifetime counter, internal voltage and internal temperature, both showing the minimum and maximum values achieved, etc.

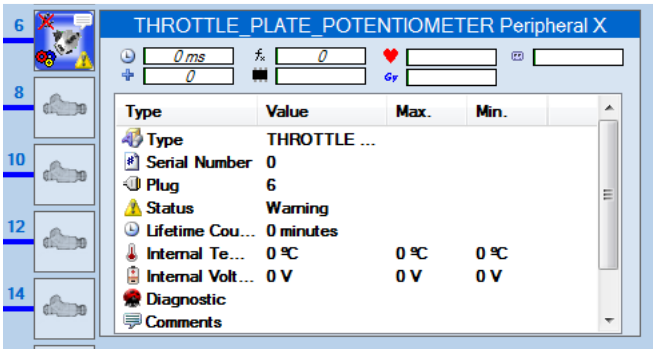

Figure 55 - Peripheral detailed information box.

When right-clicking on the icon, a menu is shown containing several options. The "*Merge Peripheral*" option is used to merge a peripheral created in the software with one plugged in the bus on the same position. This is only possible when the peripherals are from the same type or else there will be set the error status. The "*Set Lifetime*" allows for the internal lifetime counter to be reprogrammed. This is useful when testing functions or to restart the counter for remanufactured peripherals for example.

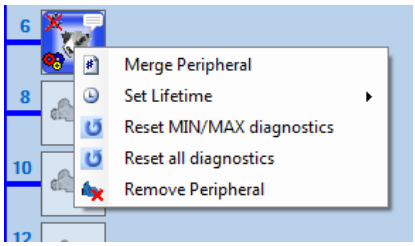

Figure 56 - Peripheral options menu.

Two other options allow the possibility to reset diagnostic variables. One resets only the maximum and minimum values stored and the other resets all internal diagnostics. Finally the last menu option removes the peripheral from the position that it has been assigned to and returns it to the "*Not Assigned Peripherals*" area.

The "*Not Assigned Peripherals*" area contains all the virtual peripherals that were created but haven't been assigned to a specific module and plug. Right clicking on the icon, a menu appears allowing the possibility to assign that particular peripheral to a specific module and plug.

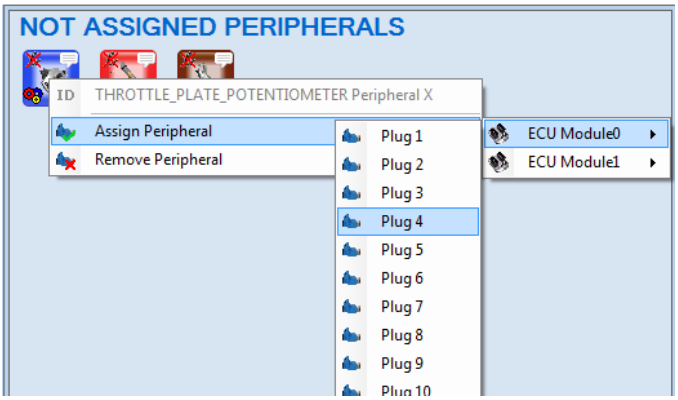

Figure 57 - *"Assign Peripheral"* menu.

The real peripherals connected in the digital bus aren't shown here as they appear in their corresponding place in the ECU architecture view.

In the remaining information box it is displayed the ECU speed and power details. The RPM speed attained by the whole system is a function of various parameters. As shown in Figure 57, it depends on the engine type (whether it is rotary or piston), the number of cylinders (up to 16) and the engine cycle (2 or 4 strokes per cycle). Depending on the selected engine properties, the maximum RPM allowed by the system varies for a given ECU setup. Changing that setup, for example adding a function, the maximum RPM also changes accordingly. The minimum desired RPM that the system must be capable of managing can be defined in the field "Desired RPM's". Having chosen that value, an alarm will be displayed to indicate if the system needs an upgrade or a revision to keep the user requirements.

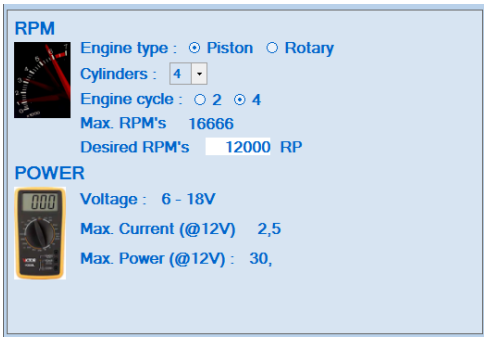

Figure 58 - System speed and power characteristics.

The power section shows an estimate of the power consumption of the entire *ECU*. The supply voltage must be in the range of 6 to 18Volts. The maximum current (thus power) consumption is estimated based on the number of modules and peripherals in the project.

On the '*Vehicle and peripherals*' tab, several views can be set up with photos of different parts of the vehicle and then position the peripheral in their equivalent physical position. In this way, it becomes easier to indentify the location of a specific peripheral if a problem occurs, also making it easier and quicker the maintenance work.

It is possible to create multiple views with more than one peripheral in each view. Peripherals can be added to the view by dragging them from the peripheral list on the left. They can also be freely moved inside de view to position them in the correct place. The zoom, pan functions and click and drag functionalities make it easier to navigate through the selected view for better positioning or location identification.

In this view, it is also possible to access the peripherals diagnostic through visual indication and more detailed in the peripheral information box if the corresponding icon is clicked.

Figure 59 - Peripheral information in a Component view.

A feature somehow similar to "Plug & Play" [19] found on computers is present in the system, here called "Plug and Run".

It is implemented in the IDMS and it works by polling, every second, each of the plugs on every available digital bus for a peripheral. Should there be a peripheral connected to a pooled plug, it will respond to that message with a message containing its unique serial number. This allows the detection of an exchanged peripheral, even if it is identical to the previous one. In case a new peripheral is detected, its internal information, including diagnostics, is requested and stored. If a peripheral does not respond to the polling from the ECU for two consecutive times (2 seconds), the peripheral is considered to be disconnected and its bus state is updated. Despite becoming disconnected, the peripheral is not automatically removed from its position. If indeed that is the desired action, it should be manually removed.

The software polls the same plug index from every connected ECU module present simultaneously.

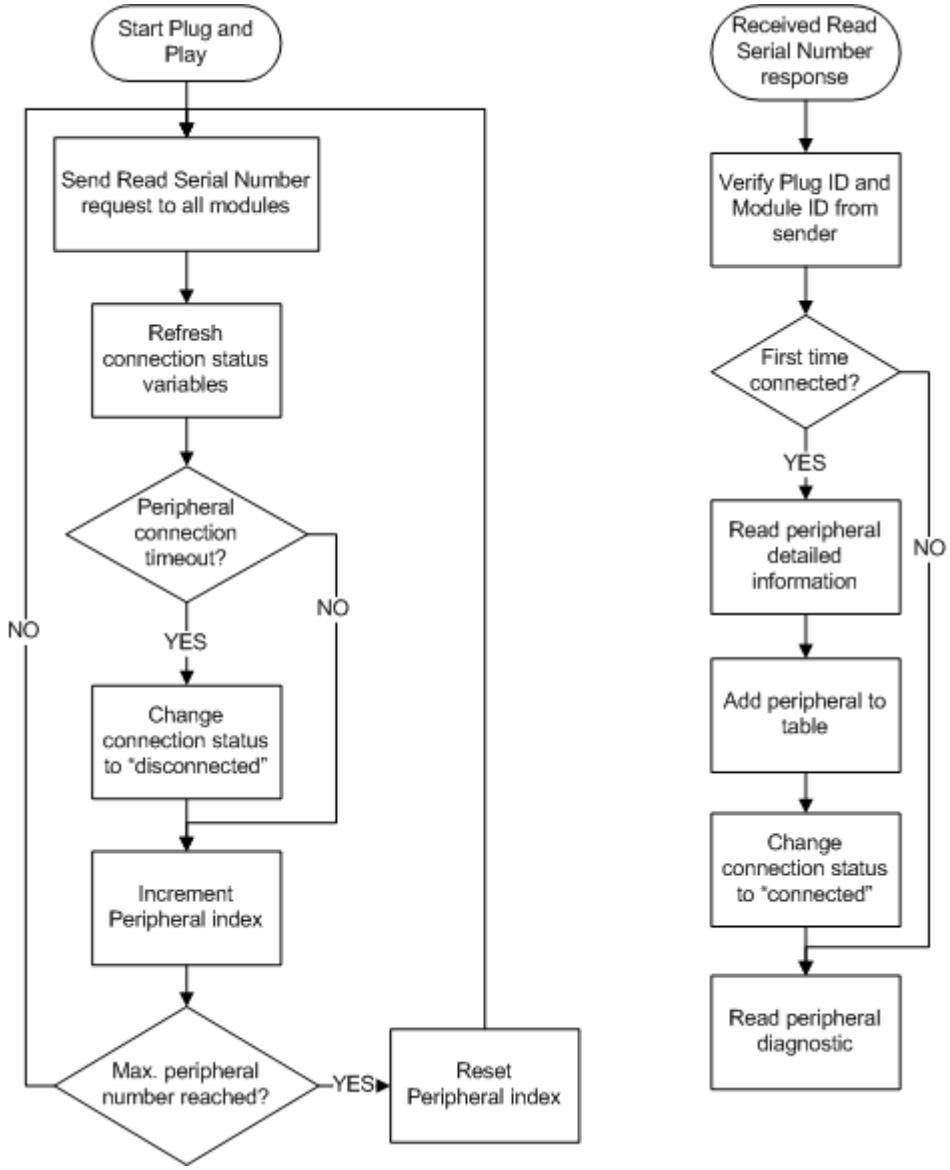

Figure 60 – "Plug and Run" functional diagram.

The "Plug and Run" operation mode is enabled in the "Communications Test" tab. It is desirable that this feature is turned off during normal system operation as it will consume peripheral bus resources unnecessarily. Normally it is only necessary during the first configuration of the system, the insertion of new peripherals or during fault detection and as it implies that a connection to a computer is established and the IDMS is running. It is not available on ECU standalone mode.

Chapter 4
Results

Summary

In this chapter it will be shown the results attained in the two main components of the project, the Digital Bus and the Intelligent Peripherals.

4.1. Digital Bus

The bus was able to meet the desired specifications with only some issues that will be presented.

Cabling:

The initial tests on the bus were made using simple phone cord wire (two pairs of untwisted wire) with a length of 5 meters and a bit rate of up to 2 Mbps were attainable between Master and Slave. When implementing the demonstration scenario with several Slaves communicating at maximum speed and the synchronization line being used simultaneously, the error rate became an issue. Only about 10% of the messages were being correctly decoded in the reception. The crosstalk effect between lines degraded the signal integrity. So the bus wiring was upgraded to coaxial 50 Ohm cable and the bit rate was also reduced to 1Mbps, allowing a higher bit decision margin that also lowered the error rate. These measures improved the message throughput to about 95%.

Error rate:

It was observed was that around 0.01% of the messages were being decoded as valid but in fact contained errors. These messages were viewed in the IDMS and it was confirmed that the data field for example, did not contain a valid BCDPFP number. This problem was due to the fact that the CRC code used had only 5 bits in length giving it low error detection capabilities.

The Synchronization line revealed stable operation in the entire RPM range used in the setup. Measures during operation, using the oscilloscope's conditional trigger, showed that no abnormal impulses were present.

Figure 61 – Digital Bus waveform. Three buses and Synchronization line(4).

4.2. Intelligent peripherals

The main goal in terms of engine control was to reach 6000 RPM with stable operation. This equals 100 crankshaft rotations per second. In a four stroke, one cylinder engine it means that, in the worst case scenario, all the control variables would have to be refreshed every 20ms (50 combustion cycles per second). But, to ensure that at least one update has occurred in a complete cycle, using an analogy of the Nyquist Law[R], the refresh frequency must be at least the double of the process frequency. This leads to a refresh rate of 10ms.

At a bit rate set to 1Mbps (750Kbps average) and an average message size of 31 bits, approximately 24.000 values are exchanged per second on a given bus. This also includes all diagnostic variables and other peripheral information. The update rate is also correlated with the number of peripherals present in a bus. In case a given bus occupancy is too high to fulfil the specifications of the system, some of the peripherals can be removed and connected to another bus with lower occupancy.

The most critical peripherals are the ones involving the spark control, such as the crankshaft angle sensor and the ignition coil. The latter must have the most up to date information from the control functions in each cycle or the engine may be damaged or even the peripheral itself. The ignition coil, and also the injector, can be damaged if, for some reason, they remain powered for a long period of time. To prevent this problem, dedicated functions were built in the code to prevent the conditions that may lead to these failures.

The crankshaft sensor, responsible for providing the synchronization pulses to all the other peripherals, can also cause engine malfunction or even destruction if it sends incorrectly formatted pulses. The code present in the auxiliary MSP from this peripheral

was kept the simplest as possible, although maintaining the required functionalities, in an effort to reduce the possibility of code, compiler or even hardware bugs.

The sensor was tested using a brushless electric motor that reached 20.000RPM. It produced valid outputs throughout the entire RPM test range.

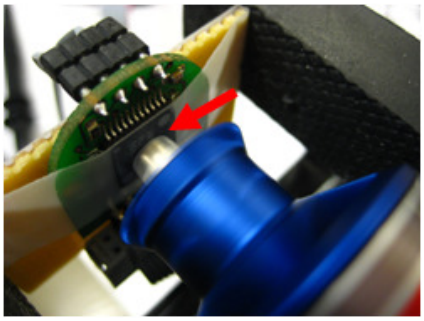

Figure 62 – Magnetic sensor test setup.

The major limitation in terms of performance is the auxiliary microcontroller that revelled to be unable to cope with that speed (20.000RPM) but enough for the RPM range of the setup engine.

The sensor is set up in the mode were it generates 800 counts per revolution. This provides an accuracy of 0.9 crankshaft degrees per count. For a maximum speed of 6000RPM, the long synchronization pulse should have a maximum of 12.5us. Measurements showed a long pulse with a period of 12us and 4.3us for the short pulse. This would lead to a maximum engine speed of 6250RPM. It was observed that, at sacrifice of accuracy on the first 6 pulses, higher speeds can be attained but it is not recommended. This is due to the fact that the MSP has 6 levels of interrupt stack so the overlapped pulses are not discarded up to that limit.

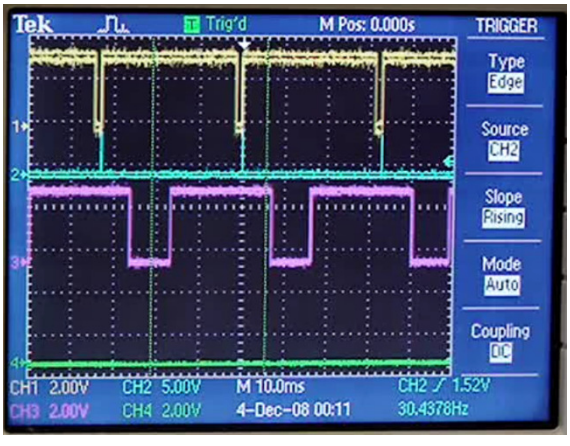

Figure 63 – Ignition(1) and injection(3) output waveforms. Also Synchronization line(2) (inverted).

Independently of the efforts in reducing the error source in the crankshaft sensor, both the ignition coil and the injector also have internal protection functions, as mentioned earlier, if, for example, too few or too many pulses are transmitted. If a more severe problem occurs with the synchronization line, like a short circuit to ground or supply voltage, that causes pulse generation to stop, inherently the engine will stop as no injections or ignitions are generated. In fact, it might happen, although the probabilities are very reduced, that an uncontrolled ignition occurs if the synchronization pulses stop during the charge cycle of the coil because there is no way of dissipating the stored energy in the coil.

4.2.1. Microcontroller

FPGA interconnection bus:

The modified SPI connection to the FPGA is able to exchange approximately 1100 variables/sec (700us variable + 220us delay). It is limited in terms of speed as it is software generated. If the FPGA block was compatible with the standard SPI hardware present in the microcontroller bit rates of up to 8Mbps could be reached. Also, the BCDPFP values are converted to and from binary values in the microcontroller for a matter of simplicity and resource saving on the FPGA side, but making the exchange of variables a slower process. Despite this, the value of 1100 variables/sec is enough to meet the requirements of 10ms refresh rate. But this depends on the number of variables that need to be exchanged. So, to help maintain these demands, the diagnostic messages are not sent all in a burst but in small groups.

Message index	Content
1	Lifetime Counter
2	Internal temperature
3	Internal voltage
4	Electric Diagnostic
5	Serial Number
6	Plug ID
7	Type

Figure 64 – Diagnostic messages index.

4.2.2. FPGA

The slew rate compensation block allowed that the effect of the inherent delays that the cable and the bus drivers introduced in the signal could be reduced to a minimum (limited in resolution by the speed of the FPGA). This allowed the bus to reach higher speeds (up to 2Mbps). The compensation can be adjusted to each device.

Due to clock signal propagation delays, the global clock speed of the core had to be reduced. This affected the available resolution on the bit decoding state machine.

The bootstrap blocked was able to bootstrap the microcontroller over the bus proving that a given peripheral could be programmed remotely even if the entire memory would to be erased.

Chapter 5
Conclusions

Summary

This chapter will focus on the conclusions related to the developed work ending with a short reference to the future work.

5.1. Digital Bus

The digital approach to peripheral interconnection to the ECU has many advantages over the analog counterpart. Only one cabling technology is used. All the connections are identical so there is no need for different wire gauges among peripherals; there is only one plug model which reduces the complexity; and the bus topology is multi-drop which means that there is no need for a dedicated connection to the ECU for each peripheral.

Functional and power delegation to the peripherals allows the analog circuitry to be confined to the peripheral itself, making it possible to have a fully digital ECU.

This architecture makes peripheral expansion an easy task. Either by adding peripherals to a bus or, if they no free plugs are available, by adding more modules to the ECU that each include a Digital Bus, thus providing more available plugs.

The synchronization line makes it possible to implement digital communications in the engine control system. The inherent delays of digital buses make them inappropriate to handle, by themselves, the demanding timing requirements of engine control.

Bus speed was enough to handle the RPM range of this engine but for Motor Sports such as Formula1, where RPM's can go up to 20.000RPM, a faster solution needs to be implemented. Also, better cabling must be used in order to extend the bus length that has been shortened in the setup.

Error detection needs to be improved. The number of CRC bits should be increased.

Due to the fact that each peripheral can be connected to any plug, it implies that power must be available in all plugs. This makes power over bus lines become unnecessary. So, bus lines can have lower voltage and less demanding current sourcing which provides higher flexibility to increase bus speed and error immunity.

5.2. Intelligent Peripherals

Having processing and storage power in the peripheral creates a wide range of possibilities. An internal lifetime counter allows the detection of imminent failure due to aging. Also storing other measurements such as temperature or voltage may also alert to imminent failure or even to determine the possible causes of any malfunction. The unique serial number prevents accidental exchange of peripherals that, even if from the same type, could pose a problem.

The values sent to the ECU are direct physical values. They do not need any processing before being used in the ECU functions. This simplifies the functions and makes the debugging easier.

Sensor calibration becomes a lesser problem with this architecture. Each sensor can be precisely calibrated in the factory and therefore be ready to use “off the shelf”. Other issues such as reading deviations due to wear or aging can be modelled and corrected in the sensor itself, removing that complexity from the ECU and the need for further adjustments when, for example, a new sensor is installed.

5.3. Software components

The “Plug and Run” operation mode simplifies the peripheral setup. It makes the addiction of new peripherals an easier task and provides a visual feedback of the peripheral connection activity.

On the “ECU” tab, the complete representation of the system allows an instantaneous location of any ECU module or peripheral problem.

Once a peripheral is connected, all the necessary information, such as internal temperature, voltage or even added comments, is easily available through the peripheral icon. Also, this is a fast method of checking if the peripheral is connected in the correct plug and in the right bus. The mini-icons provide quick visual feedback of the peripheral internal status.

The “Vehicle and Peripherals” view makes the connection between the “virtual” peripheral and its aspect and/or location in the “real life” scenario. If a problem is detected, this view allows a faster intervention.

The setup of the demonstration served as a test for the usage of the IDMS. The IDMS proved to be a useful tool to debug some of the problems that arose while assembling the whole system and also helped to solve some minor software errors regarding value conversion in the peripherals.

5.4. Future work

As an overall conclusion, the proposed objectives were successfully achieved. The main advantages and disadvantages were discussed and room for further development was found.

Future work would focus on the miniaturization of the peripheral's electronics. The modules used are not compatible with practical use and as faster and more powerful microcontrollers are being developed every day, one possible option could be having a software model of the current FPGA block running in the microcontroller lowering the systems complexity and cost. A faster Digital Bus would also be necessary as higher engine RPMs must be supported for motorsports applications to be possible.

References

[1] http://www.softing.com/home/en/industrial-automation/products/can-bus/more-can-bus/index.php?navanchor=3010320

[2] http://www.byteflight.com/homepage.htm

[3] http://www.lin-subbus.org/

[4] http://www.interfacebus.com/Design_Connector_MI_Bus.html

[5] http://www.nomadicsolutions.biz/prod.php?lg=uk&id=19

[6] http://www.smartsensorsystems.com/

[7] http://focus.ti.com/docs/prod/folders/print/msp430f1611.html

[8] http://www.massivespeedsystem.com/shopdata/0030_Focus/0020_Drivetrain/images/CM_flywheel_360x298.jpg

[9] http://www.bosch-motorsport.de/pdf/sensors/speed/IA.pdf

[10] http://www.bosch-motorsport.de/pdf/sensors/speed/HA-P.pdf

[11] http://www.rls.si/english/document/RM36D01.pdf

[12] http://www.fwicki.com/fwickis/injector2.gif

[13] http://www.bosch-motorsport.de/pdf/components/fuel_pumps/FP-100.pdf

[14] http://www.bosch-motorsport.de/pdf/components/ignition_coils/Single%20Fire%20Coil%20PS_PS_T.pdf

[15] http://www.bosch-motorsport.de/pdf/components/starters/1-4_kw.pdf

[16] http://www.bosch-motorsport.de/pdf/sensors/temperature/NTC_M12.pdf

[17] http://www.bosch-motorsport.de/pdf/sensors/air_pressure/PSB-2.pdf

[18] http://www.iar.com/website1/1.0.1.0/220/1/

[19] http://www.microsoft.com/whdc/system/pnppwr/pnp/default.mspx

Bibliography

Bierl, Lutz. *Das grosse MSP430 Praxisbuch.* 2004.
Smith, Sedra and. *Microelectronic circuits.* 1998.

Índice

1. Firmware MSP430

1.1 Main.c

```
/*********************************************************************
*******
  Filename        : Main.c

  Description     : intelligent peripherals startup code

  Project         : ECU2010
  Version         : 0.01
  Date created    : 2008-01-07
  Developer       : Rui Gomes  (maildorex@gmail.com)

*********************************************************************
*****/


#include "HeaderPer.h"

/*--- external functions of file "I2Croutines.c" ----------------------
----*/
void InitI2C(void);
void EEPROM_ByteWrite(unsigned char Address,unsigned char Data);
unsigned char EEPROM_RandomRead(unsigned char Address);
unsigned char EEPROM_CurrentAddressRead(void);
void EEPROM_AckPolling(void);
void mainI2C(void);
void wait(void);

int PtrTransmit;
unsigned char I2CBuffer[4];
unsigned char sendI2C;
unsigned char Data[128];
unsigned char data_index;
unsigned char I2C_data , I2C_address, I2C_rxdata;

//*******************************************************************
********
//        SPI variables
unsigned char spi_state, spi_frame_in_use, spi_first_read;
unsigned int spi_word_tx, spi_word_rx, spi_word_mask, spi_crc_error;
ValueStruct spi_rx_table[1];
SpiFrame spi_frame_tx, spi_frame_rx , spi_frame_aux1, spi_frame_aux2;
unsigned char spi_start_found, spi_rx_read_index, spi_rx_write_index,
spi_rx_read_index_offset;

//*******************************************************************
********
//        Global variables

unsigned char diag_mode;
unsigned int sec_counter, update_counter;
unsigned char alive_led;
unsigned char charge_state;
unsigned int diagnostic;
```

```
unsigned char diagnostic_update;
unsigned int adc_ch5, adc_ch33, adc_ch25, adc_plug, adc_vcc, adc_temp,
adc_vbat, per_plug_id;

unsigned int  start_pulse_angle, end_pulse_angle;

unsigned char button1_pressed, timer_update;

unsigned long long engine_cycle_time;

unsigned long long cycle_time_aux, cycle_time_var;

unsigned int pulse_ref_angle;

unsigned char first_rotation;
unsigned char diagnostic_enabled;
unsigned int angle_720;

//*****************************************************************
********
//     Angle related variables
unsigned int angle_offset;
unsigned int error_count;
unsigned int count_angle;

int pulse_duration;

unsigned char index;

ValueStruct data_slave[DATA_TABLE_SIZE], var, temp_sub, temp_div, var1,
var2, var3;
PerInfoStruct per_info;                         //Peripheral Information

//INFOTABLE AREA
#pragma location = 0x1000
const PerInfoStruct flash_init;


extern void MainMain(void);
extern void SendDiagState(void);

#pragma vector = RESET_VECTOR
__raw __interrupt void Main(void)
{

//*****************************************************************
*******
   // INIT STACK
   // Stack initialisation must be called directly here, because the
stack cannot handle calls yet!
   asm("MOV #0x3900, SP");

   MainMain();
}
void MainMain(void)
{
  per_info = flash_init;          // Must use variable (test) to
initialize memory
```

```
//*********************************************************************
*******
  // INIT WATCHDOG
  WDTCTL = WDTPW + WDTHOLD;
  WDTCTL = WDTPW + WDTHOLD + WDTNMIES;
  WDTCTL = WDTPW + WDTHOLD + WDTNMI + WDTNMIES;


//*********************************************************************
*******
  // INIT DCO CLOCK & XTAL
  P2DIR_bit.P2DIR_5 = PIN_INPUT;
  P2SEL_bit.P2SEL_5 = TRUE;
  BCSCTL2_bit.DCOR = TRUE;
  BCSCTL2_bit.SELS = FALSE;
  DCOCTL_bit.DCO0 = TRUE;
  DCOCTL_bit.DCO1 = TRUE;
  DCOCTL_bit.DCO2 = TRUE;
  BCSCTL1_bit.RSEL0 = TRUE;
  BCSCTL1_bit.RSEL1 = TRUE;
  BCSCTL1_bit.RSEL2 = TRUE;

  // Disable unneeded clock circuits (XT2)
  BCSCTL1_bit.XT2OFF = TRUE;

  // Disable unneeded clock circuits (LFXT1)
  asm("bis #0x0020, SR");  // asm("bis #OSCOFF, SR");



//*********************************************************************
*******
  //INIT ADC
  // Initialise A/D converter

  P6SEL = 15;                          //disable CMOS gates
  ADC12CTL0_bit.ADC12TOVIE = FALSE;  //disable timer overflow int
  ADC12CTL0_bit.ADC12OVIE = FALSE;   //disable overflow int
  ADC12CTL0_bit.REFON = TRUE;      // ADC12 Reference on
  ADC12CTL0_bit.REF2_5V = TRUE;     // ADC12 Reference 2,5V
  ADC12CTL0_bit.MSC = TRUE;        // ADC12 Multiple SampleConversion on
  ADC12CTL0_bit.SHT0= 15;           // ADC12 Sample Hold 0    1024 cycles
  ADC12CTL1_bit.CONSEQ = 1;        // ADC12 Conversion Sequence Select
multiple channel
  ADC12CTL1_bit.ADC12SSEL = 2;     // ADC12 Clock Source Select MCLK
  ADC12CTL1_bit.ADC12DIV = 3;      // ADC12 Clock Divider Select  =1
  ADC12CTL1_bit.SHS =0;            // ADC12 Sample/Hold Source
  ADC12CTL1_bit.SHP =1;
  ADC12CTL1_bit.CSTARTADD = 0;     // ADC12 Conversion Start Address
  ADC12MCTL0_bit.INCH = 3;
  ADC12MCTL0_bit.SREF = 1;  // for 5V
  ADC12MCTL0_bit.EOS = 0;
  ADC12MCTL1_bit.INCH = 4;
  ADC12MCTL1_bit.SREF = 0;  // for 3,3V
  ADC12MCTL1_bit.EOS = 0;
  ADC12MCTL2_bit.INCH = 5;
  ADC12MCTL2_bit.SREF = 1;  // for 2,5V
  ADC12MCTL2_bit.EOS = 0;
```

```
  ADC12MCTL3_bit.INCH = 6;        //VBATT
  ADC12MCTL3_bit.SREF = 1;
  ADC12MCTL3_bit.EOS = 0;
  ADC12MCTL4_bit.INCH = 2;        //PLUG ID
  ADC12MCTL4_bit.SREF = 0;
  ADC12MCTL4_bit.EOS = 0;
  ADC12MCTL5_bit.INCH = 10;      //Internal temperature
  ADC12MCTL5_bit.SREF = 1;
  ADC12MCTL5_bit.EOS = 0;
  ADC12MCTL6_bit.INCH = 11;       // Vcc/2
  ADC12MCTL6_bit.SREF = 1;
  ADC12MCTL6_bit.EOS = 1;         //  last channel

  // Enable SMCLK output to pin P1.4
  //P1SEL_bit.P1SEL_4 = FALSE;
  //P1DIR_bit.P1DIR_4 = PIN_OUTPUT;


//*********************************************************************
*******
  // INIT TIMERS
  // Initialise timer A (0.1 seconds per interrupt call)
  TACTL_bit.TASSEL = 2;             // 2:SMCLK
  TACTL_bit.TAID = 3;               // 3: DIV = 8
  TACTL_bit.TAMC = 2;               // 3: UP/DOWN
  TACCR1 = 50000;
  TACCR2 = 0xFFFF;

  TACCTL2_bit.CCIE = FALSE;
  TACCTL2_bit.CCIFG = FALSE;
  TACCTL1_bit.CCIE = TRUE;
  TACCTL1_bit.CCIFG = FALSE;
  TACTL_bit.TAIE = TRUE;



//*********************************************************************
*******
  // PIN CONFIGURATION
  PIN_ALIVELED_DIR = PIN_OUTPUT;
  PIN_ALIVELED_OUT = TRUE;
  PIN_DEBUGLED_DIR = PIN_OUTPUT;
  PIN_DEBUGLED_OUT = FALSE;
  PIN_OUTLED_DIR = PIN_OUTPUT;
  PIN_OUTLED_OUT = FALSE;

  PIN_SPI_READY_DIR = PIN_INPUT;

  PIN_DIGITAL_INPUT_1_DIR = PIN_INPUT;
  PIN_DIGITAL_INPUT_2_DIR = PIN_INPUT;
  PIN_DIGITAL_INPUT_3_DIR = PIN_INPUT;
  PIN_DIGITAL_INPUT_4_DIR = PIN_INPUT;

  PIN_DIGITAL_OUTPUT_1_DIR = PIN_OUTPUT;
  PIN_DIGITAL_OUTPUT_2_DIR = PIN_OUTPUT;
  PIN_DIGITAL_OUTPUT_3_DIR = PIN_OUTPUT;
  PIN_DIGITAL_OUTPUT_4_DIR = PIN_OUTPUT;

  PIN_DIGITAL_OUTPUT_1 = TRUE;
```

```
  PIN_DIGITAL_OUTPUT_2 = FALSE;
  PIN_DIGITAL_OUTPUT_3 = FALSE;
  PIN_DIGITAL_OUTPUT_4 = FALSE;

  PIN_POWER_DIR = PIN_OUTPUT;
  PIN_POWER_OUT = POWER_OFF;

  PIN_SPI_CLK_DIR = PIN_OUTPUT;
  PIN_SPI_CLK_OUT = FALSE;
  PIN_SPI_SIMO_DIR = PIN_OUTPUT;
  PIN_SPI_SIMO_OUT = FALSE;
  PIN_SPI_SEND_DIR = PIN_OUTPUT;
  PIN_SPI_SEND_OUT = FALSE;
  PIN_SPI_SOMI_DIR = PIN_INPUT;
  PIN_SPI_SLAVE_READY_DIR = PIN_INPUT;



//*********************************************************************
*******
  // INIT GLOBAL VARIABLES
//#define PER_IGNITION_COIL                 23
//#define PER_MAGNETIC_FUEL_INJECTION_VALVE 15
//#define PER_GENERIC_IN_OUT                91
//#define PER_INTAKE_CAMSHAFT_POTENTIOMETER 10
//#define PER_LED_LAMP                      44
//#define PER_FUEL_PRESSURE_SENSOR          12
//#define PER_FUEL_PUMP                     19
//#define PER_AIRBOX_PRESSURE_SENSOR        2
//#define PER_INTAKE_AIR_TEMPERATURE_SENSOR 4
//#define PER_OIL_TEMPERATURE_SENSOR        52
//#define PER_STARTER                       81
//#define PER_THROTTLE_PLATE_POTENTIOMETER  6
//#define PER_CRANKSHAFT_INDUCTIVE_SENSOR   37
//#define PER_BATTERY_MONITOR_SENSOR        92

  per_info = flash_init;                  // Must use variable (test) to
initialize memory
  if(per_info.lifetime_count == 0)       // set default values
  {
    per_info.serial_number = 13231;
    per_info.per_type = PER_INTAKE_AIR_TEMPERATURE_SENSOR;
    per_info.firmware_version = 0x01;      //Firmware Version

    per_info.min_int_temp = 0xFFFF;
    per_info.max_int_temp = 0;
    per_info.min_vcc = 0xFFFF;
    per_info.max_vcc = 0;
    per_info.mag_angle_offset = 0;
  }

  if((per_info.per_type == PER_MAGNETIC_FUEL_INJECTION_VALVE) ||
(per_info.per_type == PER_IGNITION_COIL))
  {
    Init_SPI();
  }
  if(per_info.per_type == PER_CRANKSHAFT_INDUCTIVE_SENSOR)
  {
```

```
  // Initialise timer B
  TBCTL_bit.TBSSEL = 2;             // 2:SMCLK
  TBCTL_bit.TBID = 3;               // 0: DIV = 1 //3: DIV = 8
  TBCTL_bit.TBMC = 2;               // 1: UP 2: CONTINUOUS
  TBCCR0 = 0xFFFF;

  TBCCTL0_bit.CCIE = TRUE;
  TBCCTL0_bit.CCIFG = FALSE;
  TBCTL_bit.TBIE = TRUE;

  Init_SPI();
  PIN_TDC_DIR = PIN_INPUT;
  PIN_TDC_IES = FALSE;
  PIN_TDC_IE = TRUE;
}
if((per_info.per_type == PER_THROTTLE_PLATE_POTENTIOMETER) ||
   (per_info.per_type == PER_OIL_TEMPERATURE_SENSOR) ||
   (per_info.per_type == PER_INTAKE_AIR_TEMPERATURE_SENSOR) ||
   (per_info.per_type == PER_FUEL_PRESSURE_SENSOR) ||
   (per_info.per_type == PER_AIRBOX_PRESSURE_SENSOR))
{
  diagnostic_enabled = 1;
}
else
{
  diagnostic_enabled = 0;
}
angle_offset = per_info.mag_angle_offset;
spi_start_found = FALSE;
spi_first_read = FALSE;
spi_frame_in_use = 0;
spi_crc_error = 0;

alive_led = 0;
sec_counter = 0;
charge_state = 0;

diagnostic_update = FALSE;
diagnostic = 0;

index = 0;

adc_ch5 = 0, adc_ch33 = 0, adc_ch25 = 0;
adc_temp = 0, adc_vcc = 0;
adc_vbat = 0;

button1_pressed = FALSE;
timer_update = FALSE;
start_pulse_angle = 0;
end_pulse_angle = 0;

error_count = 0;
count_angle = 0;

cycle_time_aux = 0;
cycle_time_var = 999999;

first_rotation = 1;
```

```
  unsigned int i=0;
  unsigned char diag_interval = 0;

 //I2C
//  InitI2C();    // Initialize I2C module
//  Init_SPI();   // Initialize SPI module


//*********************************************************************
*******
  // ENABLE GLOBAL INTERRUPTS
  GIE_Set;

   //TEST
  //IE1_bit.NMIIE = TRUE;      //enable NMI interrupt


//*********************************************************************
*******
  // MAIN CYCLE

  //generate_crc_table(0x008D);
    //I2C
//  while(1)
//  {
//    if(diagnostic_update)
//    {
//      diagnostic_update = FALSE;
//      mainI2C();
//    }
//  };

  Read_Adc(1);
  Read_Adc(1);

  for(;;)
  {
    Read_Adc(0);
    switch(per_info.per_type)
    {
      case PER_AIRBOX_PRESSURE_SENSOR:
        SPI_Send_Data(OUT_ADC1,adc_ch5,1000);
        break;
      case PER_FUEL_PRESSURE_SENSOR:
        SPI_Send_Data(OUT_ADC1,adc_ch5,1000);
        break;
      case PER_INTAKE_AIR_TEMPERATURE_SENSOR:
        SPI_Send_Data(OUT_ADC1,adc_ch33,1000);
        break;
      case PER_OIL_TEMPERATURE_SENSOR:
        SPI_Send_Data(OUT_ADC1,adc_ch33,1000);
        break;
      case PER_THROTTLE_PLATE_POTENTIOMETER:
        SPI_Send_Data(OUT_ADC1,adc_ch25,1000);
        break;
      case PER_GENERIC_IN_OUT:
        SPI_Send_Data(OUT_ADC1,adc_ch33,1000);
        SPI_Send_Data(OUT_DIGITAL_INPUT_1,!PIN_DIGITAL_INPUT_1,1);
        SPI_Send_Data(OUT_DIGITAL_INPUT_2,!PIN_DIGITAL_INPUT_2,1);
```

```
          break;
        case PER_BATTERY_MONITOR_SENSOR:
          SPI_Send_Data(OUT_ADC1,adc_vbat,1000);
          SPI_Send_Data(OUT_VCC,adc_vbat,1000);
          break;
        case PER_MAGNETIC_FUEL_INJECTION_VALVE:
          SPI_Send_Data(OUT_VCC,adc_vbat,1000);
          break;
        case PER_IGNITION_COIL:
          SPI_Send_Data(OUT_VCC,adc_vbat,1000);
          break;
        case PER_CRANKSHAFT_INDUCTIVE_SENSOR:
              if(cycle_time_var == 999999)
              {
                // 1111 0100 0010 0011 1111  = 0x00FF423F  -> infinite
                SPI_Send_Data(OUT_ENGINE_RAW_SPEED,0x00FF423F,0);
              }
              else if(cycle_time_var > 7500)
                {

SPI_Send_Data(OUT_ENGINE_RAW_SPEED,cycle_time_var,981000);
//(1.000.000)981000 Actual MSP clock period = 127ns
              }
              //SPI_Send_Data(OUT_VCC,adc_vbat,1000);
              angle_720 = count_angle;
              if(angle_720 > 800)
                angle_720 -= 800;
              angle_720 = angle_720*9/10;
              SPI_Send_Data(OUT_ANGLE,angle_720,1);
              SPI_Send_Data(OUT_ANGLE_ERROR_COUNT,error_count,1);
          break;
      }
      SPI_Send_Data(800,i++,1);

      if(diagnostic_update)
      {
        if(spi_comms_enabled == 0)
        {
          if(PIN_SPI_READY_IN == 1)
          {
            U1IE += UTXIE1;
            spi_comms_enabled = 1;
          }
        }
        else
        {
          if(PIN_SPI_READY_IN == 0)
          {
            IFG2_bit.UTXIFG1 = 1;
            spi_comms_enabled = 1;
          }
        }
        diagnostic_update = FALSE;
        Read_Adc(1);

        diagnostic = 0;

        switch(per_info.per_type)
        {
```

```
case PER_AIRBOX_PRESSURE_SENSOR:
    if(adc_ch5 < DIAG_SHORT_5_LOW_LEVEL)
    {
      diagnostic = diagnostic + DIAG_SHORT_GND;
    }
    else
      if(adc_ch5 > DIAG_SHORT_5_HIGH_LEVEL)
      {
        diagnostic = diagnostic + DIAG_SHORT_VCC;
      }
  break;
case PER_FUEL_PRESSURE_SENSOR:
    if(adc_ch5 < DIAG_SHORT_5_LOW_LEVEL)
    {
      diagnostic = diagnostic + DIAG_SHORT_GND;
    }
    else
      if(adc_ch5 > DIAG_SHORT_5_HIGH_LEVEL)
      {
        diagnostic = diagnostic + DIAG_SHORT_VCC;
      }
  break;
case PER_INTAKE_AIR_TEMPERATURE_SENSOR:
    if(adc_ch33 < DIAG_SHORT_33_LOW_LEVEL)
    {
      diagnostic = diagnostic + DIAG_SHORT_GND;
    }
    else
      if(adc_ch33 > DIAG_SHORT_33_HIGH_LEVEL)
      {
        diagnostic = diagnostic + DIAG_SHORT_VCC;
      }
  break;
case PER_OIL_TEMPERATURE_SENSOR:
    if(adc_ch33 < DIAG_SHORT_33_LOW_LEVEL)
    {
      diagnostic = diagnostic + DIAG_SHORT_GND;
    }
    else
      if(adc_ch33 > DIAG_SHORT_33_HIGH_LEVEL)
      {
        diagnostic = diagnostic + DIAG_SHORT_VCC;
      }
  break;
case PER_THROTTLE_PLATE_POTENTIOMETER:
    if(adc_ch33 < DIAG_SHORT_33_LOW_LEVEL)
    {
      diagnostic = diagnostic + DIAG_SHORT_GND;
    }
    else
      if(adc_ch33 > DIAG_SHORT_33_HIGH_LEVEL)
      {
        diagnostic = diagnostic + DIAG_SHORT_VCC;
      }
  break;
}

SPI_Send_Data(OUT_TEMP,adc_temp,10);
if(diagnostic_enabled)
```

```
      {
        SPI_Send_Data(OUT_DIAG,diagnostic,1);
      }
      else
      {
        SPI_Send_Data(OUT_DIAG,0,1);
      }
      switch(diag_interval++)
      {
        case 1 :
          SPI_Send_Data(OUT_LIFETIME, per_info.lifetime_count,1);
          SPI_Send_Data(OUT_PLUGID_BCDP,per_plug_id,1);
          SPI_Send_Data(OUT_PLUGID_BIN,per_plug_id,0);
          break;
        case 2 :
          SPI_Send_Data(OUT_DIAG_MIN_TEMP,per_info.min_int_temp,10);
          SPI_Send_Data(OUT_DIAG_MAX_TEMP,per_info.max_int_temp,10);
          SPI_Send_Data(OUT_DIAG_MIN_VCC,per_info.min_vcc,1000);
          break;
        case 3 :
          SPI_Send_Data(OUT_DIAG_MAX_VCC,per_info.max_vcc,1000);
          SPI_Send_Data(OUT_SERIAL,per_info.serial_number,1);
          SPI_Send_Data(OUT_TYPE,per_info.per_type,1);
          diag_interval = 0;
          break;
      }
      // *********************************
      // LIFE_TIME COUNTER
      if(sec_counter >= 600)          //sec. default=60   5=test
      {
        sec_counter = 0;
        per_info.lifetime_count++;
        //Write Info Flash
        Write_Info(per_info.raw, 0, 128);
      }
    }


  }
}
```

1.2 Int.c

```
/*****************************************************************************
*******
  Filename        : Int.c

  Description     : Interrupt routines

  Project         : ECU2010
  Version         : 0.01
  Date created    : 2008-01-07
  Developer's     : Rui Gomes  (maildorex@gmail.com)

*****************************************************************************
*****/
```

```c
#include "HeaderPer.h"

/*-------------------------------------------------------------------------
----*/
/* Interrupt Service Routines */
/* Note that the Compiler version is checked in the following code and */
/* depending of the Compiler Version the correct Interrupt Service */
/* Routine definition is used. */

unsigned int ij;

// Description:
// Byte Write Operation. The communication via the I2C bus with an EEPROM

// Description:
// Byte Write Operation. The communication via the I2C bus with an EEPROM
#pragma vector=USART0TX_VECTOR
__interrupt void ISR_I2C(void)
{
      switch (I2CIV)
      { case I2CIV_AL: /* I2C interrupt vector: Arbitration lost (ALIFG)
*/
            break;
      case I2CIV_NACK: /* I2C interrupt vector: No acknowledge (NACKIFG)
*/
            break;
      case I2CIV_OA: /* I2C interrupt vector: Own address (OAIFG) */
            break;
      case I2CIV_ARDY: /* I2C interrupt vector: Access ready (ARDYIFG) */
            break;
      case I2CIV_RXRDY: /* I2C interrupt vector: Receive ready (RXRDYIFG)
*/
            I2C_rxdata = I2CDRB; // store received data in buffer
                //data_index++;
            break;
      case I2CIV_TXRDY: /* I2C interrupt vector: Transmit ready
(TXRDYIFG) */
                  I2CDRB =  I2CBuffer[PtrTransmit];
                  PtrTransmit = PtrTransmit - 1;
                  if (PtrTransmit<0)
                  {
                        I2CIE &= ~TXRDYIE; // disable interrupts
                        I2CTCTL |= I2CSTP; // stop condition is
generated after slave address was sent
                        //I2CIFG_bit.TXRDYIFG = 0;
                  }
                  for(ij = 0; ij < 1500; ij++);
            break;
      case I2CIV_GC: /* I2C interrupt vector: General call (GCIFG) */
            break;
      case I2CIV_STT: /* I2C interrupt vector: Start condition (STTIFG)
*/
            break;
      }
}
```

```
//*********************************************************************
********
// TIMER A
// Realtime timing generation (... seconds per interrupt call)

long auxvar2;
#pragma vector = TIMERA1_VECTOR
__interrupt void TimerA1_Interrupt(void)
{
   TACTL_bit.TAIFG = FALSE;

   if(TACCTL1_bit.CCIFG && TACCTL1_bit.CCIE)
   {
     auxvar2 = TAR + 50000;
     TACCTL1_bit.CCIFG = FALSE;

     if(auxvar2 > 0xFFFF)
     {
       auxvar2 -= 0xFFFF;
     }

     TACCR1 = auxvar2;

      //*********************************
      // LEDS
      // Blink internal "alive" for program OK (short-blink)
      ++alive_led;
      if(alive_led >=20)
      {
         PIN_ALIVELED_OUT = TRUE;
         ++sec_counter;
         if(sec_counter > 1)
         {
           diagnostic_update = TRUE;
         }
         alive_led = 0;
         PIN_OUTLED_OUT =  FALSE;
      }
      else
      {
        PIN_ALIVELED_OUT = FALSE;
      }
   }
}

//*********************************************************************
********
// PORT2
// TDC interrupt service routine
#pragma vector = PORT2_VECTOR
__interrupt void Port2_Interrupt(void)
{
  P2IFG = 0;
  if(first_rotation)
  {
    first_rotation = 0;
    cycle_time_var = 999999;
  }
  else
```

```
  {
    cycle_time_var = cycle_time_aux + TBR;
  }
  cycle_time_aux = 0;
  TBR = 0;
}

//*********************************************************************
********
// TIMER B
// Cycle time counter

#pragma vector = TIMERB0_VECTOR
__interrupt void TimerB0_Interrupt(void)
{
  TBCTL_bit.TBIFG = FALSE;
  TBCCTL0_bit.CCIFG = FALSE;
  cycle_time_aux += 0x10000;
  if((cycle_time_aux >> 16) > 15)
  {
    cycle_time_aux = 0;
    cycle_time_var = 999999;
    first_rotation = 1;
  }
}
```

1.3 Functions.c

```
/*********************************************************************
*******
  Filename        : Functions.c

  Description     : Communications routines

  Project         : ECU2010
  Version         : 0.01
  Date created    : 2008-01-07
  Developer's     : Rui Gomes  (maildorex@gmail.com)

*********************************************************************
*****/

#include "HeaderPer.h"

/*
const static unsigned long factor_2500div4095 = 0x7950C4;      // =
0.610500
//const static unsigned long factor_3500div4095 = 0x7D0AAC;      // =
0.854700
const static unsigned long factor_3300div4095 = 0x7C4BE4;      // =
0.805860
const static unsigned long factor_5000div4095 = 0x81DCF4;      // =
1.22100
const static unsigned long factor_temp_offset = 0x7F0B90;      // = 0.986
```

```
const static unsigned long factor_temp_gain   = 0x556AB8;      // =
0.00355
const static unsigned long factor_1000        = 0xB186A0;      // = 1000
const static unsigned long factor_500         = 0xA7A120;      // = 500
*/
const char crc8_Table[256] =
{
0x00 ,0x8D ,0x97 ,0x1A ,0xA3 ,0x2E ,0x34 ,0xB9 ,0xCB ,0x46 ,0x5C ,0xD1
,0x68 ,0xE5 ,0xFF ,0x72,
0x1B ,0x96 ,0x8C ,0x01 ,0xB8 ,0x35 ,0x2F ,0xA2 ,0xD0 ,0x5D ,0x47 ,0xCA
,0x73 ,0xFE ,0xE4 ,0x69,
0x36 ,0xBB ,0xA1 ,0x2C ,0x95 ,0x18 ,0x02 ,0x8F ,0xFD ,0x70 ,0x6A ,0xE7
,0x5E ,0xD3 ,0xC9 ,0x44,
0x2D ,0xA0 ,0xBA ,0x37 ,0x8E ,0x03 ,0x19 ,0x94 ,0xE6 ,0x6B ,0x71 ,0xFC
,0x45 ,0xC8 ,0xD2 ,0x5F,
0x6C ,0xE1 ,0xFB ,0x76 ,0xCF ,0x42 ,0x58 ,0xD5 ,0xA7 ,0x2A ,0x30 ,0xBD
,0x04 ,0x89 ,0x93 ,0x1E,
0x77 ,0xFA ,0xE0 ,0x6D ,0xD4 ,0x59 ,0x43 ,0xCE ,0xBC ,0x31 ,0x2B ,0xA6
,0x1F ,0x92 ,0x88 ,0x05,
0x5A ,0xD7 ,0xCD ,0x40 ,0xF9 ,0x74 ,0x6E ,0xE3 ,0x91 ,0x1C ,0x06 ,0x8B
,0x32 ,0xBF ,0xA5 ,0x28,
0x41 ,0xCC ,0xD6 ,0x5B ,0xE2 ,0x6F ,0x75 ,0xF8 ,0x8A ,0x07 ,0x1D ,0x90
,0x29 ,0xA4 ,0xBE ,0x33,
0xD8 ,0x55 ,0x4F ,0xC2 ,0x7B ,0xF6 ,0xEC ,0x61 ,0x13 ,0x9E ,0x84 ,0x09
,0xB0 ,0x3D ,0x27 ,0xAA,
0xC3 ,0x4E ,0x54 ,0xD9 ,0x60 ,0xED ,0xF7 ,0x7A ,0x08 ,0x85 ,0x9F ,0x12
,0xAB ,0x26 ,0x3C ,0xB1,
0xEE ,0x63 ,0x79 ,0xF4 ,0x4D ,0xC0 ,0xDA ,0x57 ,0x25 ,0xA8 ,0xB2 ,0x3F
,0x86 ,0x0B ,0x11 ,0x9C,
0xF5 ,0x78 ,0x62 ,0xEF ,0x56 ,0xDB ,0xC1 ,0x4C ,0x3E ,0xB3 ,0xA9 ,0x24
,0x9D ,0x10 ,0x0A ,0x87,
0xB4 ,0x39 ,0x23 ,0xAE ,0x17 ,0x9A ,0x80 ,0x0D ,0x7F ,0xF2 ,0xE8 ,0x65
,0xDC ,0x51 ,0x4B ,0xC6,
0xAF ,0x22 ,0x38 ,0xB5 ,0x0C ,0x81 ,0x9B ,0x16 ,0x64 ,0xE9 ,0xF3 ,0x7E
,0xC7 ,0x4A ,0x50 ,0xDD,
0x82 ,0x0F ,0x15 ,0x98 ,0x21 ,0xAC ,0xB6 ,0x3B ,0x49 ,0xC4 ,0xDE ,0x53
,0xEA ,0x67 ,0x7D ,0xF0,
0x99 ,0x14 ,0x0E ,0x83 ,0x3A ,0xB7 ,0xAD ,0x20 ,0x52 ,0xDF ,0xC5 ,0x48
,0xF1 ,0x7C ,0x66 ,0xEB
};

//*****************************************************************
********
// read ADC values

void Read_Adc(unsigned char enable_diagnostic)
{

    ADC12CTL0_bit.ADC12ON = TRUE;       // Enable ADC
    ADC12CTL0_bit.ENC = TRUE;

    ADC12CTL0_bit.ADC12SC = TRUE;      // start sampling
    ADC12CTL0_bit.ADC12SC = FALSE;     // start conversion
    adc_ch5 = ((unsigned long)ADC12MEM0)*5000/4095;  //500 clocks

    ADC12CTL0_bit.ADC12SC = TRUE;      // start sampling
    ADC12CTL0_bit.ADC12SC = FALSE;     // start conversion
    adc_ch33 = ((unsigned long)ADC12MEM1)*3300/4095;
```

```
    ADC12CTL0_bit.ADC12SC = TRUE;      // start sampling
    ADC12CTL0_bit.ADC12SC = FALSE;     // start conversion
    adc_ch25 = ((unsigned long)ADC12MEM2)*2500/4095;

    ADC12CTL0_bit.ADC12SC = TRUE;      // start sampling
    ADC12CTL0_bit.ADC12SC = FALSE;     // start conversion
//    adc_vbat = ((unsigned long)ADC12MEM3)*17813/4095;  //2500/4095;

    if(enable_diagnostic)
    {
      ADC12CTL0_bit.ADC12SC = TRUE;      // start sampling
      ADC12CTL0_bit.ADC12SC = FALSE;     // start conversion
      if(ADC12MEM4 < 4050)
      {
        per_plug_id = ((unsigned long)ADC12MEM4)*16/4095;
      }
      else
      {
        per_plug_id = 0;
      }

      ADC12CTL0_bit.ADC12SC = TRUE;      // start sampling
      ADC12CTL0_bit.ADC12SC = FALSE;     // start conversion
      adc_temp = ((unsigned long)ADC12MEM5)*25000/4095;    //ADC12MEM4
      adc_temp = (((unsigned long)adc_temp)*100-986000)/355;

      ADC12CTL0_bit.ADC12SC = TRUE;      // start sampling
      ADC12CTL0_bit.ADC12SC = FALSE;     // start conversion
      adc_vcc = ((unsigned long)ADC12MEM6)*5000/4095;

      if((adc_temp > per_info.max_int_temp) && (adc_temp < 1000))
      {
        per_info.max_int_temp = adc_temp;
      }
      if((adc_temp < per_info.min_int_temp) && (adc_temp > 0))
      {
        per_info.min_int_temp = adc_temp;
      }

      if((adc_vcc > per_info.max_vcc) && (adc_vcc < 5000))
      {
        per_info.max_vcc = adc_vcc;
      }
      if((adc_vcc < per_info.min_vcc) && (adc_vcc > 2000))
      {
        per_info.min_vcc = adc_vcc;
      }
    }

    ADC12CTL0_bit.ADC12ON = FALSE;    //Disable ADC
    ADC12CTL0_bit.ENC = FALSE;

    return;
}

//*********************************************************************
********
// Send SPI data frame
unsigned char crc_value;
```

```
unsigned int aux_addr;
unsigned char i, sizes;
long aux_var;

void SPI_Send_Data(unsigned int spi_data_addr, unsigned long
spi_data_value, unsigned long div_factor)
{
  spi_frame_tx.start = 0x2A;
  spi_frame_tx.cmd = 0x01;
  spi_frame_tx.addr = spi_data_addr;
  var1.bcdp_value = Long32ToBCDP((signed long)spi_data_value);
  if(div_factor != 1)
  {
    if(div_factor == 0)
    {
      var1.raw = (unsigned long long)spi_data_value;
    }
    else
    {
      var1.bcdp_value = BCDPFPdiv(var1.bcdp_value , Long32ToBCDP((signed
long)div_factor));   //800 clocks aprox
    }
  }

  spi_frame_tx.value = var1.rawBcdpfpSign;
  spi_frame_tx.stop = 0x15;

  crc_value = 0x00;
  crc_value = crc8_Table[crc_value ^ (spi_frame_tx.byte_array[6] &
0x07)];
  crc_value = crc8_Table[crc_value ^ spi_frame_tx.byte_array[5]];
  crc_value = crc8_Table[crc_value ^ spi_frame_tx.byte_array[4]];
  crc_value = crc8_Table[crc_value ^ spi_frame_tx.byte_array[3]];
  crc_value = crc8_Table[crc_value ^ spi_frame_tx.byte_array[2]];
  crc_value = crc8_Table[crc_value ^ spi_frame_tx.byte_array[1]];
  spi_frame_tx.crc = crc_value; //

  if(spi_frame_in_use == 0)
  {
    spi_frame_aux1.word1 = SPI_Comm_Word(spi_frame_tx.word1);
    spi_frame_aux1.word2 = SPI_Comm_Word(spi_frame_tx.word2);
    spi_frame_aux1.word3 = SPI_Comm_Word(spi_frame_tx.word3);
    spi_frame_aux1.word4 = SPI_Comm_Word(spi_frame_tx.word4);
  }
  else
  {
    spi_frame_aux2.word1 = SPI_Comm_Word(spi_frame_tx.word1);
    spi_frame_aux2.word2 = SPI_Comm_Word(spi_frame_tx.word2);
    spi_frame_aux2.word3 = SPI_Comm_Word(spi_frame_tx.word3);
    spi_frame_aux2.word4 = SPI_Comm_Word(spi_frame_tx.word4);
  }

  if(spi_start_found == FALSE)
  {
    i= 0;
    spi_first_read = TRUE;
    if(spi_frame_in_use == 0)
    {
      while(i<8)
```

```
      {
        if(spi_frame_aux1.byte_array[i] == 0x2A)
        {
          spi_start_found = TRUE;
          spi_rx_read_index_offset = i;
          spi_rx_write_index = 0;
        }
        i++;
      }
    }
    else
    {
      while(i<8)
      {
        if(spi_frame_aux2.byte_array[i] == 0x2A)
        {
          spi_start_found = TRUE;
          spi_rx_read_index_offset = i;
          spi_rx_write_index = 0;
        }
        i++;
      }
    }
  }
  else
  {
    if(spi_rx_read_index_offset != 0)
    {
      if(spi_frame_in_use == 0)
      {
        spi_rx_read_index = spi_rx_read_index_offset;
        spi_rx_write_index = 0;
        while(spi_rx_read_index < 8)
        {
          spi_frame_rx.byte_array[spi_rx_write_index++] =
spi_frame_aux2.byte_array[spi_rx_read_index++];
        }
        spi_rx_read_index = 0;
        while(spi_rx_read_index < spi_rx_read_index_offset)
        {
          spi_frame_rx.byte_array[spi_rx_write_index++] =
spi_frame_aux1.byte_array[spi_rx_read_index++];
        }
      }
      else
      {
        spi_rx_read_index = spi_rx_read_index_offset;
        spi_rx_write_index = 0;
        while(spi_rx_read_index < 8)
        {
          spi_frame_rx.byte_array[spi_rx_write_index++] =
spi_frame_aux1.byte_array[spi_rx_read_index++];
        }
        spi_rx_read_index = 0;
        while(spi_rx_read_index < spi_rx_read_index_offset)
        {
          spi_frame_rx.byte_array[spi_rx_write_index++] =
spi_frame_aux2.byte_array[spi_rx_read_index++];
```

```
        }
      }
    }
    else
    {
      if(spi_frame_in_use == 0)
      {
        spi_rx_read_index = spi_rx_read_index_offset;
        spi_rx_write_index = 0;
        while(spi_rx_read_index < 8)
        {
          spi_frame_rx.byte_array[spi_rx_write_index++] =
spi_frame_aux1.byte_array[spi_rx_read_index++];
        }
        spi_rx_read_index = 0;
        while(spi_rx_read_index < spi_rx_read_index_offset)
        {
          spi_frame_rx.byte_array[spi_rx_write_index++] =
spi_frame_aux2.byte_array[spi_rx_read_index++];
        }

      }
      else
      {
        spi_rx_read_index = spi_rx_read_index_offset;
        spi_rx_write_index = 0;
        while(spi_rx_read_index < 8)
        {
          spi_frame_rx.byte_array[spi_rx_write_index++] =
spi_frame_aux2.byte_array[spi_rx_read_index++];
        }
        spi_rx_read_index = 0;
        while(spi_rx_read_index < spi_rx_read_index_offset)
        {
          spi_frame_rx.byte_array[spi_rx_write_index++] =
spi_frame_aux1.byte_array[spi_rx_read_index++];
        }
      }
    }
    if(spi_frame_rx.byte_array[0] == 0x2A)
    {
      crc_value = 0x00;
      crc_value = crc8_Table[crc_value ^ (spi_frame_rx.byte_array[6] &
0x07)];
      crc_value = crc8_Table[crc_value ^ spi_frame_rx.byte_array[5]];
      crc_value = crc8_Table[crc_value ^ spi_frame_rx.byte_array[4]];
      crc_value = crc8_Table[crc_value ^ spi_frame_rx.byte_array[3]];
      crc_value = crc8_Table[crc_value ^ spi_frame_rx.byte_array[2]];
      crc_value = crc8_Table[crc_value ^ spi_frame_rx.byte_array[1]];

      if(crc_value == spi_frame_rx.crc)
      {
        PIN_OUTLED_OUT = TRUE;

        spi_rx_table[0].rawBcdpfpSign = spi_frame_rx.value;

        aux_var = BCDPFPToLong32(spi_rx_table[0].bcdp_value);
        switch(spi_frame_rx.addr)
        {
```

```
          case IN_PULSE_OPEN_ANGLE :
            if(per_info.per_type == PER_MAGNETIC_FUEL_INJECTION_VALVE)
            {
              start_pulse_angle = (unsigned int)(800 -
(((int)aux_var*10)/9));
              if(start_pulse_angle >= 800)
              {
                start_pulse_angle = start_pulse_angle - 800;
              }
            }
            else
            {
              start_pulse_angle = (unsigned int)(800 -
(((int)aux_var*10)/9));
              if(start_pulse_angle >= 800)
              {
                start_pulse_angle = start_pulse_angle - 800;
              }
            }
            break;

          case IN_PULSE_CLOSE_ANGLE :
            if(per_info.per_type == PER_MAGNETIC_FUEL_INJECTION_VALVE)
            {
              end_pulse_angle = (unsigned int)aux_var;
            }
            else
            {
              end_pulse_angle = (unsigned int)(800 -
(((int)aux_var*10)/9));
              if(end_pulse_angle > 800)
              {
                end_pulse_angle = end_pulse_angle - 800;
              }
            }
            break;

          case IN_DIGITAL_OUTPUT_1 :
            PIN_DIGITAL_OUTPUT_1 = !((unsigned char)aux_var && 0x01);
            PIN_OUTLED_OUT =  TRUE;
            break;

          case IN_DIGITAL_OUTPUT_2 :
            PIN_DIGITAL_OUTPUT_2 = (unsigned char)aux_var && 0x01;
            break;

          case IN_DIGITAL_OUTPUT_3 :
            PIN_DIGITAL_OUTPUT_3 = (unsigned char)aux_var && 0x01;
            break;

         case IN_DIGITAL_OUTPUT_4 :
            PIN_DIGITAL_OUTPUT_4 = (unsigned char)aux_var && 0x01;
            break;

          case IN_CONFIG :
            if(aux_var & CONFIG_TYPE_MASK)
            {
              per_info.per_type = aux_var & 0xFFFF;
              //Write_Info(per_info.raw, 0, 128);
```

```
            }
            else
            {
              if(aux_var & CONFIG_SERIAL_MASK)
              {
                per_info.serial_number = (per_info.serial_number &
0xFFFF0000) | (aux_var & 0xFFFF);
                //Write_Info(per_info.raw, 0, 128);
              }
              else
              {
                if(aux_var & CONFIG_LIFETIME_MASK)
                {
                  per_info.serial_number = (per_info.lifetime_count &
0xFFFF0000) | (aux_var & 0xFFFF);
                  //Write_Info(per_info.raw, 0, 128);
                }
               else
                {
                  if(aux_var & CONFIG_FW_OFFSET_MASK)
                  {
                    per_info.mag_angle_offset = (unsigned int)((unsigned
int)aux_var*10/9);
                    angle_offset = per_info.mag_angle_offset;
                    Write_Info(per_info.raw, 0, 128);
                  }
                }
              }
            }
            break;

          case IN_ERASE_INFO :
            if(aux_var == ERASE_ALL_PASSWORD)
            {
              for(i = 0; i<128; i++)
              {
                per_info.raw[i] = 0;
              }
              per_info.min_int_temp = 0xFFFF;
              per_info.min_vcc = 0xFFFF;
              Write_Info(per_info.raw, 0, 128);
              PIN_OUTLED_OUT = TRUE;
              //clear config
            }
            else
            {
              if(aux_var == ERASE_DIAG_PASSWORD)
              {
                per_info.diagnostic = 0;
                per_info.short_gnd = 0;
                per_info.short_vcc = 0;
                per_info.mag_angle_offset = 0;
                per_info.min_int_temp = 0xFFFF;
                per_info.max_int_temp = 0;
                per_info.min_vcc = 0xFFFF;
                per_info.max_vcc = 0;
                Write_Info(per_info.raw, 0, 128);
                PIN_OUTLED_OUT = TRUE;
                //clear config
```

```
                }
                else
                {
                  if(aux_var == ERASE_MINMAX_PASSWORD)
                  {
                    per_info.min_int_temp = 0xFFFF;
                    per_info.max_int_temp = 0;
                    per_info.min_vcc = 0xFFFF;
                    per_info.max_vcc = 0;
                    Write_Info(per_info.raw, 0, 128);
                    PIN_OUTLED_OUT = TRUE;
                    //clear config
                  }
                  else
                  {
                    PIN_OUTLED_OUT = FALSE;
                  }
                }
              }
              break;
          }
        }
        else
        {
          var2.rawBcdpfpSign = spi_frame_rx.value;
          spi_crc_error++;
        }
      }
      else
      {
        spi_start_found = FALSE;
        spi_rx_write_index = 0;
      }
    }
    spi_frame_in_use = ! spi_frame_in_use;



}
 /*
unsigned char crctab[256];

void generate_crc_table(unsigned long polynom)
{

      // make CRC lookup table used by table algorithms

      int i, j;
      unsigned long bit, crc;//, polynom;
//        polynom = 0x008C;

      for (i=0; i<256; i++)
        {
            crc=(unsigned long)i;

            for (j=0; j<8; j++)
                {
                  bit = crc & 0x80;
                  crc<<= 1;
```

```
                if (bit) crc^= polynom;
            }

            crc&= 0xFF;
            crctab[i]= crc;
        }
        crc = 0;
}

*/
unsigned char spi_bit_counter;
unsigned char wait_timeout;

unsigned int SPI_Comm_Word(unsigned int spi_word_tx)
{
  spi_word_mask = 0x8000;
  PIN_SPI_CLK_OUT = 0;
  spi_state = 1;
  spi_bit_counter = 0;
  spi_word_rx = 0;
  wait_timeout = 250;
  while(!PIN_SPI_SLAVE_READY_IN && wait_timeout)
  {
    wait_timeout --;
  }
  if(wait_timeout > 0)
  {
    PIN_DEBUGLED_OUT = 1;
    PIN_SPI_SEND_OUT = 1;
    while(spi_bit_counter < 16)
    {
      if(spi_state == 1)
       {

         if((spi_word_tx & spi_word_mask) != 0)
         {
            PIN_SPI_SIMO_OUT = 1;
         }
         else
         {
            PIN_SPI_SIMO_OUT = 0;
         }
         PIN_SPI_CLK_OUT = 1;
       }
       else
       {
         PIN_SPI_CLK_OUT = 0;
         if(PIN_SPI_SOMI_IN)
         {
            spi_word_rx |= spi_word_mask;
         }
         spi_word_mask = spi_word_mask >> 1;
         spi_bit_counter ++;
         spi_state = 0;
       }
      spi_state++;
    }
    spi_bit_counter = 0;
    PIN_SPI_SEND_OUT = 0;
```

```
    PIN_DEBUGLED_OUT = 0;
  }
  return spi_word_rx;
}
```

1.4 SPI.c

```
/*********************************************************************
*******
  Filename         : SPI.c

  Description      : SPI Interrupt routines

  Project          : ECU2010
  Version          : 0.01
  Date created     : 2008-11-10
  Developer's      : Rui Gomes  (maildorex@gmail.com)

*********************************************************************
*****/

#include "HeaderPer.h"


//-------------------------------------------------------------------
-------
// SPI HARDWARE variables
SPI_FRAME spi_tx_frame;
SPI_FRAME spi_rx_frame;
unsigned char spi_rx_index, spi_tx_index;
unsigned char spi_rx_sync;
unsigned char spi_comms_enabled;

unsigned char aux_byte;

#ifdef SPIMASTER
//-------------------------------------------------------------------
-------
// SPI Master


//*******************************************************************
*******
  // SPI Init
  void Init_SPI(void)
  {
    /*
      1) Set SWRST (BIS.B #SWRST,&UxCTL)
      2) Initialize all USART registers with SWRST=1 (including UxCTL)
      3) Enable USART module via the MEx SFRs (USPIEx)
      4) Clear SWRST via software (BIC.B #SWRST,&UxCTL)
      5) Enable interrupts (optional) via the IEx SFRs (URXIEx and/or
UTXIEx)
    */
    //asm("BIS.B #SWRST,&UxCTL");
```

```
  U1CTL = SWRST;
  U1CTL += CHAR + SYNC + MM;
  U1TCTL = SSEL0 + SSEL1 + STC;
  U1MCTL = 0;
#ifdef INDUCTIVE
    U1BR0 = 5;
    U1BR1 = 5;
#else
    U1BR0 = 40;
    U1BR1 = 0;
#endif
  U1ME = USPIE1;
  //asm("BIC.B #SWRST,&UxCTL");
  U1CTL &= ~SWRST;
  U1IE += URXIE1; //          U0IE += UTXIE0 + URXIE0;

  PIN_SPI_SIMO_SEL = TRUE;
  PIN_SPI_SOMI_SEL = TRUE;
  PIN_SPI_CLK_SEL = TRUE;

  spi_tx_frame.raw[0] = 0xAA;
  spi_tx_frame.raw[5] = 0x55;
  spi_tx_frame.start_angle = 0;
  spi_tx_frame.end_angle = 0;

  spi_rx_index = 0;
  spi_tx_index = 0;
  spi_rx_sync = 0;
  spi_comms_enabled = 0;
}
//---------------------------------------------------------------------
---------
// SPI Master interrupts

#pragma vector = UART1TX_VECTOR
__interrupt void SPI_TX_Interrupt(void)
{
  U1TXBUF = spi_tx_frame.raw[spi_tx_index];

  spi_tx_index++;
  if(spi_tx_index > 5)
  {
    spi_tx_index = 0;

    #ifdef INDUCTIVE
        spi_tx_frame.start_angle = angle_offset;  //inductive
    #else
        spi_tx_frame.start_angle = start_pulse_angle;
        spi_tx_frame.end_angle = end_pulse_angle;
    #endif

  }

}

#pragma vector = UART1RX_VECTOR
__interrupt void SPI_RX_Interrupt(void)
{
  aux_byte = U1RXBUF;
```

```
    if(spi_rx_sync == 0)
    {
      if(aux_byte == 0xAA)
      {
        spi_rx_sync = 1;
        spi_rx_index = 0;
      }
    }
    if(spi_rx_sync == 1)
    {
      spi_rx_frame.raw[spi_rx_index] = aux_byte;
      spi_rx_index++;
      if(spi_rx_index > 5)
      {
        spi_rx_index = 0;
        if((spi_rx_frame.raw[0] == 0xAA) && (spi_rx_frame.raw[5] ==
0x55))
        {
         #ifdef INDUCTIVE
              error_count = spi_rx_frame.raw[1];
              error_count |= (unsigned int)spi_rx_frame.raw[2] << 8;
              count_angle = spi_rx_frame.raw[3];
              count_angle |= ((unsigned int)spi_rx_frame.raw[4]) << 8;
          #else
              engine_cycle_time = spi_rx_frame.cycle_time;
          #endif

        }
        else
        {
          spi_rx_sync = 0;
        }
      }
    }
  }

#else

#endif
```


2. VHDL Code

1.5 Peripheral Controller

```
----------------------------------------------------------------------------
---------
-- Company:
```

```
-- Engineer: Rui Gomes
--
-- Create Date:    15:12:37 06/12/2007
-- Design Name:
-- Module Name:    perController - Behavioral
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------
---------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.NUMERIC_STD.ALL;


entity peripheralController is
    Port (
                clk : in  std_logic;
                rst : in  std_logic;
                device_is_bus_master : in  std_logic;

                pin_rx                  : in std_logic;
                pin_tx                  : out std_logic;
                pin_crank_in      : in std_logic;
                pin_crank_out     : out std_logic;

                tx_cmd            : in std_logic_vector(3 downto 0);
                tx_data           : in std_logic_vector(24 downto 0);
                tx_address  : in std_logic_vector(19 downto 0);
                tx_data_msg_size        : in std_logic_vector(3 downto
0);
                tx_data_frame_size      : in std_logic_vector(10 downto
0);

                rx_cmd            : out std_logic_vector(3 downto 0);
                rx_data           : out std_logic_vector(24 downto 0);
                rx_address  : out std_logic_vector(19 downto 0);



                per_interface_in  : in std_logic_vector(4 downto 0);
                                  -- tx_start rx_ack bootstrap_mode
bootstrap_en
                per_interface_out : out std_logic_vector(5 downto 0);
                                  -- tx_ack rx_ready bootstrap_en_ack
s_rx_receiving_data s_tx_sending_data
                per_interface_bus_speed : in std_logic_vector(7 downto
0);
```

```
                debug_tx    : out std_logic_vector(15 downto 0);
                debug_rx    : out std_logic_vector(29 downto 0);
                switchper   : in std_logic_vector(0 downto 0);
                led_out           : out std_logic_vector(7 downto 0);

                -------------------------
                -- Data Memory interface
                tx_addr_mem : out std_logic_vector(3 downto 0);
                tx_din_mem  : in std_logic_vector(7 downto 0);

                rx_addr_mem : out std_logic_vector(3 downto 0);
                rx_we_mem   : out std_logic;
                rx_dout_mem : out std_logic_vector(7 downto 0)

        );
end peripheralController;

architecture Behavioral of peripheralController is

  -- For message CMD field
   constant CMD_DATA_FRAME     : std_logic_vector(1 downto 0) := "00";
   constant CMD_LIVE_PROTOTYPE : std_logic_vector(1 downto 0) := "01";
   constant CMD_BROADCAST      : std_logic_vector(1 downto 0) := "10";
   constant CMD_GIMY_ACCESS    : std_logic_vector(1 downto 0) := "11";

   -- For message "direction" field
   constant CMD_REQUEST_USB  : std_logic := '0';
   constant CMD_RESPONSE_USB : std_logic := '1';

   -- For read/write field
   constant CMD_READ_VALUE   : std_logic := '0';
   constant CMD_WRITE_VALUE  : std_logic := '1';

      --For peripheral interface signal fields
      constant IN_TX_START                              : integer := 0;
      constant IN_RX_ACK                                : integer := 1;
      constant IN_BOOTSTRAP_MODE                : integer := 2;
      constant IN_BOOTSTRAP_EN                  : integer := 3;
      constant IN_BOOTSTRAP_NEXT_FRAME : integer := 4;

      constant OUT_TX_ACK                               : integer := 0;
      constant OUT_RX_READY                     : integer := 1;
      constant OUT_BOOTSTRAP_EN_ACK       : integer := 2;
      constant OUT_BOOTSTRAP_FRAME_SENT: integer := 3;
      constant OUT_RX_RECEIVING_DATA      : integer := 4;
      constant OUT_TX_SENDING_DATA        : integer := 5;

-----------------------------------------------------------------------
--RX
            type rx_state_type is (st_rx_sync, st_rx_cmd, st_rx_addr8,
st_rx_addr15, st_rx_addr20,

      st_rx_bcdp_data, st_rx_data_frame, st_rx_dummy, st_rx_crc);
            signal rx_state : rx_state_type;

            signal rx_data_byte                               :
std_logic_vector(7 downto 0);
```

```
            signal s_rx_cmd                                 :
std_logic_vector(3 downto 0);
            signal s_rx_calculate_data_crc      : std_logic;
            signal rx_data_byte_size                : std_logic_vector(3
downto 0);
            signal rx_data_type                             : std_logic;
            signal count                                        :
std_logic_vector(15 downto 0);
            signal rx_byte_count                    : natural range 0 to
15;

            signal rx_bit_error                             : natural
range  0 to 60;
            signal rx_timer                                 : natural
range 0 to 8191;
            signal rx_one_bit_time                  : natural range 0 to
8191;
            signal rx_zero_bit_time                 : natural range 0 to
8191;
            signal rx_bit_timeout                   : natural range 0 to
8191;
            signal rx_bit_time_tolerance        : natural range 0 to 8191;
            signal old_pin_rx                               : std_logic ;

            signal rx_error_count                   :
std_logic_vector(20 downto 0);
            signal s_frame_complete                 : std_logic;
            signal s_rx_calculate_crc               : std_logic;
            signal rx_bit_count                             : natural
range 0 to 30;
            signal odd_pins                                 : natural
range 0 to 3;
            signal s_rx_receiving_data          : std_logic;
            signal s_rx_data                                :
std_logic_vector(24 downto 0);
            signal s_rx_address                             :
std_logic_vector(19 downto 0);

            signal rx_ack                                       :
std_logic;
            signal rx_ready                                 : std_logic;

-------------------------------------------------------------------
--CRC
            signal tx_crc                   : std_logic_vector(4 downto 0);
            signal rx_calc_crc      : std_logic_vector(4 downto 0);
            signal rx_crc_var       : std_logic_vector(4 downto 0);
-------------------------------------------------------------------
--TX
            type tx_state_type is (st_tx_sync, st_tx_cmd, st_tx_addr8,
st_tx_addr15, st_tx_addr20,
                                                        st_tx_dummy,
st_tx_data, st_tx_data_frame, st_tx_crc, st_tx_end_delay,
st_tx_end_delay_2, st_tx_end);
            signal tx_state : tx_state_type;


            signal tx_data_byte                             : std_logic_vector(7
downto 0);
```

```
--          signal s_tx_cmd                         : std_logic_vector(3
downto 0);
            signal s_tx_address                     :
std_logic_vector(19 downto 0);
            signal s_tx_data_frame_size   : std_logic_vector(3 downto 0);
            signal s_tx_calculate_crc           : std_logic;

            signal output_state           : std_logic_vector(1 downto 0);
            signal tx_bit_count           : natural range 0 to 256;
            signal tx_byte_count    : natural range 0 to 15;

            signal tx_data_type           : std_logic;

            signal s_tx_sending_data      : std_logic;
            signal per_tx_enable          : std_logic;
            signal s_pin_tx_aux                 : std_logic;
            signal pin_aux_b                    : std_logic;
            signal s_pin_tx                     : std_logic;
            signal pin_tx_b                     : std_logic;
            signal pin_oldaux_a                 : std_logic;
            signal pin_oldaux_b                 : std_logic;

            signal tx_bit_time                  : natural range 0 to
65535;
            signal tx_counter                   : natural range 0 to
65535;
            signal tx_one_bit_time        : natural range 0 to 8191;
            signal tx_zero_bit_time       : natural range 0 to 8191;
            signal tx_bootst_bit_time     : natural range 0 to 8191;

            signal mosfet_a_state         : natural range 0 to 255;
            signal tx_start                     : std_logic;
            signal tx_ack                             : std_logic;

-----------------------------------------------------------------------
--Debug
            signal aux8       : std_logic_vector(7 downto 0);
-----------------------------------------------------------------------
--Bootstrap
            type bootRamType        is array (0 to 6) of
std_logic_vector(7 downto 0);
            type bootFrameType      is array (0 to 3) of bootRamType;
            signal boot_cmd_frame         : bootFrameType ;

            type boot_state_type is (boot_idle, boot_start,
boot_send_byte, boot_send_sync, boot_send_cmd, boot_ack_wait, boot_end);
            signal boot_state                   : boot_state_type;

            signal boot_bit_counter       : natural range 0 to 31;
            signal timer                          : natural range 0 to
2000;
            signal boot_start_vector_a : std_logic_vector(10 downto 0);
            signal boot_start_vector_b : std_logic_vector(10 downto 0);
            signal boot_data                    : std_logic_vector(7
downto 0);
            signal boot_ckl                     : std_logic_vector(7
downto 0);
            signal boot_ckh                     : std_logic_vector(7
downto 0);
```

```
            signal boot_data_flash        : std_logic_vector(7 downto 0);
            signal boot_data_parity       : std_logic;
            signal boot_ck_bit                  : std_logic;
            signal boot_sync_en_bit       : std_logic;
            signal boot_frame_number      : natural range 0 to 31;
            signal boot_frame_size        : natural range 0 to 127;
            signal boot_byte_counter      : natural range 0 to 127;
            signal wait_counter                 : natural range 0 to 254;

            signal bootstrap_mode               : std_logic;
            signal bootstrap_en                       : std_logic;
            signal bootstrap_en_ack             : std_logic;
            signal bootstrap_frame_sent   : std_logic;
            signal bootstrap_next_frame   : std_logic;

-------------------------------------------------------
--MEM interface
            type TX_MEM_ST_type is(st_idle,
st_write_tx_mem_0,st_write_tx_mem_1 );
            signal st_tx_mem : TX_MEM_ST_type;

            type RX_MEM_ST_type is(st_idle, st_write_rx_mem );
            signal st_rx_mem : RX_MEM_ST_type;

            signal s_tx_addr_mem          : std_logic_vector(3 downto 0);
            signal s_write_byte_count     : std_logic_vector(3 downto 0);
            signal s_write_data_mem       : std_logic;
            signal s_rx_addr_mem          : std_logic_vector(3 downto 0);

begin

      tx_start          <= per_interface_in(IN_TX_START);
      rx_ack                  <= per_interface_in(IN_RX_ACK);
      bootstrap_mode <= per_interface_in(IN_BOOTSTRAP_MODE) ;
      bootstrap_en      <= per_interface_in(IN_BOOTSTRAP_EN);
      bootstrap_next_frame <= per_interface_in(IN_BOOTSTRAP_NEXT_FRAME);

      per_interface_out(OUT_TX_ACK)                                 <=
tx_ack;
      per_interface_out(OUT_RX_READY)                               <=
rx_ready;
      per_interface_out(OUT_BOOTSTRAP_EN_ACK)         <= bootstrap_en_ack;

      per_interface_out(OUT_BOOTSTRAP_FRAME_SENT)     <=
bootstrap_frame_sent;
      per_interface_out(OUT_RX_RECEIVING_DATA)        <=
s_rx_receiving_data;
      per_interface_out(OUT_TX_SENDING_DATA)          <=
s_tx_sending_data;

      aux8 <= per_interface_bus_speed(7 downto 0);

      tx_addr_mem <= s_tx_addr_mem;

      rx_cmd <= s_rx_cmd;
      rx_addr_mem <= s_rx_addr_mem;

      -------------------------------------------------
      ------    Begin of TX State Machine  -----------
```

```
	-----------------------------------------------

	-- Clock Divider
	process(clk, rst)
	begin
		if (rst = '1') then
			count<= (others => '0');
		elsif rising_edge( clk ) then
			count <= count + 1;    --  count for delay
		end if;
	end process;

	debug_tx(4 downto 0) <= tx_crc(4 downto 0);

	-- Combinatorial part of the state machine
	process(rst, clk, output_state, s_pin_tx)
			variable output_ctrl_en : std_logic;
	begin


		if (rst = '1') then
			s_pin_tx_aux <= '1';
			pin_crank_out <= '1';
			pin_oldaux_a <= '1';
			pin_oldaux_b <= '1';
			output_ctrl_en := '1';

			tx_counter <= 0;
			tx_byte_count <= 0;
			tx_bit_count <= 0;
			tx_bit_time <= 2;
			tx_one_bit_time <= 20;              -- 60 *20ns  ,  16 *
80ns
			tx_zero_bit_time <= 10;             -- 30 *20ns ,  8 *
80ns
			tx_bootst_bit_time <= 5208;         -- *20ns


			output_state <= "01";
			per_tx_enable <= '0';
			s_tx_sending_data <= '0';
			tx_ack <= '0';
			tx_state <= st_tx_sync;
			s_tx_calculate_crc <= '0';

			boot_state <= boot_start;
			boot_bit_counter <= 0;
			timer <= 0;
			boot_start_vector_b <= "11100000000";
			boot_start_vector_a <= "11001101111";
			boot_data <= x"80";
			boot_cmd_frame(0) <= (x"18", x"04", x"04", x"00",
x"00", x"00", x"00");         --Mass erase
			boot_cmd_frame(1) <= (x"10", x"24", x"24", x"00",
x"00", x"00", x"00");         --Send Password
			boot_cmd_frame(2) <= (x"12", x"24", x"24", x"00",
x"40", x"20", x"00");         --Write Flash
			boot_cmd_frame(3) <= (x"14", x"24", x"24", x"00",
x"40", x"20", x"00");         --Read Flash
```

```
                        boot_frame_size <= 6;
                        boot_data_parity <= '0';
                        boot_bit_counter <= 0;
                        bootstrap_en_ack <= '0';
                        boot_data_flash <= x"00";
                        bootstrap_frame_sent <= '0';
                        tx_data_byte <=  x"00";

                elsif (rising_edge(clk) and (s_rx_receiving_data = '0')) then
--                              case switchper(3 downto 2) is
--                                    when "00" =>
--                                          tx_one_bit_time <= 60;
--                                          tx_zero_bit_time <= 30;
--                                    when "01" =>
--                                          tx_one_bit_time <= 40;
--                                          tx_zero_bit_time <= 20;
--                                    when "10" =>
--                                          tx_one_bit_time <= 30;
--                                          tx_zero_bit_time <= 15;
--                                    when "11" =>
--                                          tx_one_bit_time <= 20;
--                                          tx_zero_bit_time <= 10;
--                                    when others =>
--                              end case;

                              tx_counter <= tx_counter + 1;
                              if(tx_counter = tx_bit_time) then
                                    tx_counter <= 0;
                                    if(per_tx_enable = '1') then
                                          tx_bit_count <= tx_bit_count + 1;
                                          if(bootstrap_mode = '0') then
                        ----------------------------------------
                        --TX normal operation
                        ----------------------------------------
                                                      case tx_state is
                                                            when st_tx_sync =>
                                                                  case tx_bit_count is
                                                                        when 0 =>

      output_ctrl_en := '1';

      tx_one_bit_time <= conv_integer("0000" & per_interface_bus_speed(7
downto 0) & '0');

      tx_zero_bit_time <= conv_integer("00000" &
per_interface_bus_speed(7 downto 0));

      s_tx_sending_data <= '1';

      tx_bit_time <= conv_integer("0000" & per_interface_bus_speed(7
downto 0) & '0');             --long delay (sync bits)

      s_tx_address(19 downto 0) <= tx_address(19 downto 0);
                                                                                    tx_crc
<= "11111";
                                                                  when 1 =>

      tx_bit_time <= tx_zero_bit_time;               --short delay (sync
bits)
```

```
      tx_bit_count <= 0;
                                                                  tx_state
<= st_tx_cmd;
                                                              when others =>

      tx_bit_count <= 0;
                                                          end case;

                                                    when st_tx_cmd =>

      if(tx_cmd(tx_bit_count) = '1') then

                                                          tx_bit_time <=
tx_one_bit_time;
                                                          else
                                                          tx_bit_time <=
tx_zero_bit_time;
                                                          end if;

                                                          case tx_bit_count is
                                                              when 0 =>
                                                              when 1 =>

      if(tx_cmd(1 downto 0) = CMD_DATA_FRAME) then

      tx_state <= st_tx_addr20;                 --20 bit address (Data
Frame)

      tx_bit_count <= 0;

      tx_byte_count <= 0;
                                                                  end if;

                                                              when 2 =>
                                                                  if
(tx_cmd(1 downto 0) = CMD_BROADCAST) then

      tx_state <= st_tx_addr20;                 --20 bit address
(Broadcast)

      tx_bit_count <= 0;

      tx_byte_count <= 0;
                                                                  elsif
(tx_cmd(1 downto 0) = CMD_GIMY_ACCESS) then

      tx_state <= st_tx_addr8;                  --8 bit address (Gimy
normal access)

      tx_bit_count <= 0;

      tx_byte_count <= 0;
                                                                  end if;

                                                              when 3 =>
                                                                  tx_state
<= st_tx_addr15;              --15 bit address (Gimy direct addressing)
```

```
      tx_bit_count <= 0;
      tx_byte_count <= 0;
                                                      when others =>
      tx_bit_count <= 0;
                                                  end case;

                                              when st_tx_addr8 =>
      if(s_tx_address(tx_bit_count) = '1') then  --address 8 bits Routing
Table
                                                      tx_bit_time <=
tx_one_bit_time;
                                                  else
                                                      tx_bit_time <=
tx_zero_bit_time;
                                                  end if;

                                                  if(tx_bit_count = 7)
then
      s_tx_address(19 downto 8) <= (others => '0');
                                                      tx_bit_count
<= 0;
                                                      if (tx_cmd(2)
= CMD_WRITE_VALUE) then
                                                          tx_state
<= st_tx_data;
                                                      else
                                                          tx_state
<= st_tx_dummy;
                                                      end if;
                                                  end if;

                                              when st_tx_addr15 =>
      if(s_tx_address(tx_bit_count) = '1') then--address 15 bits Gimy
Direct Access
                                                      tx_bit_time <=
tx_one_bit_time;
                                                  else
                                                      tx_bit_time <=
tx_zero_bit_time;
                                                  end if;

                                                  if(tx_bit_count =
14) then
      s_tx_address(19 downto 15) <= (others => '0');
                                                      if ((tx_cmd(2)
= CMD_RESPONSE_USB) or (tx_cmd(3) = CMD_WRITE_VALUE)) then
                                                          tx_state
<= st_tx_data;
                                                      else
                                                          tx_state
<= st_tx_dummy;
```

```
                                                                    end if;
                                                                    tx_bit_count
<= 0;
                                                                end if;

                                                            when st_tx_addr20 =>

      if(s_tx_address(tx_bit_count) = '1') then--address 20 bits
Broadcast
                                                                    tx_bit_time <=
tx_one_bit_time;
                                                                else
                                                                    tx_bit_time <=
tx_zero_bit_time;
                                                                end if;
                                                                if(tx_bit_count =
19) then
                                                                    tx_bit_count
<= 0;
                                                                    if(tx_cmd(1
downto 0) = CMD_DATA_FRAME) then
                                                                        tx_state
<= st_tx_data_frame;                --4 bit address (PlugID)

      s_tx_data_frame_size <= tx_address(7 downto 4);

      s_tx_addr_mem<= (others => '0');

      tx_data_byte <=  tx_din_mem;

      tx_bit_count <= 0;

      tx_byte_count <= 0;
                                                                    else
                                                                        if
(tx_cmd(2) = CMD_WRITE_VALUE) then

      tx_state <= st_tx_data;
                                                                        else

      tx_state <= st_tx_dummy;
                                                                        end if;
                                                                    end if;
                                                                end if;

                                                            when st_tx_data =>

      if(tx_data(tx_bit_count) = '1') then          --data bits
                                                                    tx_bit_time <=
tx_one_bit_time;
                                                                else
                                                                    tx_bit_time <=
tx_zero_bit_time;
                                                                end if;
                                                                if(tx_bit_count =
24) then
                                                                    tx_bit_count
<= 0;
```

```
                                                        -- polynomial:
(0 1 2 3 5) 0x2F
                                                        --data width:
49
                                                        --calculate
CRC5
                                                        tx_crc(0) <=
tx_cmd(3) xor tx_cmd(2) xor tx_cmd(1) xor tx_data(22) xor tx_data(19) xor
tx_data(18) xor tx_data(17) xor tx_data(16) xor tx_data(15) xor
tx_data(13) xor

tx_data(12) xor tx_data(11) xor tx_data(7) xor tx_data(5) xor tx_data(3)
xor tx_data(2) xor tx_data(0) xor

s_tx_address(19) xor s_tx_address(18) xor s_tx_address(17) xor
s_tx_address(13) xor s_tx_address(11) xor

s_tx_address(9) xor s_tx_address(8) xor s_tx_address(6) xor
s_tx_address(1) xor s_tx_address(0);
                                                        tx_crc(1) <=
tx_cmd(1) xor tx_cmd(0) xor tx_data(23) xor tx_data(22) xor tx_data(20)
xor tx_data(15) xor tx_data(14) xor

tx_data(11) xor tx_data(8) xor tx_data(7) xor tx_data(6) xor tx_data(5)
xor tx_data(4) xor tx_data(2) xor tx_data(1) xor tx_data(0) xor

s_tx_address(17) xor s_tx_address(14) xor s_tx_address(13) xor
s_tx_address(12) xor

s_tx_address(11) xor s_tx_address(10) xor s_tx_address(8) xor
s_tx_address(7) xor s_tx_address(6) xor s_tx_address(2) xor
s_tx_address(0);
                                                        tx_crc(2) <=
tx_cmd(3) xor tx_data(24) xor tx_data(23) xor tx_data(22) xor

tx_data(21) xor tx_data(19) xor tx_data(18) xor tx_data(17) xor
tx_data(13) xor tx_data(11) xor

tx_data(9) xor tx_data(8) xor tx_data(6) xor tx_data(1) xor tx_data(0)
xor s_tx_address(19) xor s_tx_address(17) xor s_tx_address(15) xor

s_tx_address(14) xor s_tx_address(12) xor s_tx_address(7) xor
s_tx_address(6) xor s_tx_address(3) xor s_tx_address(0);
                                                        tx_crc(3) <=
tx_cmd(3) xor tx_cmd(2) xor tx_cmd(1) xor tx_cmd(0) xor tx_data(24) xor
tx_data(23) xor tx_data(20) xor tx_data(17) xor tx_data(16) xor
tx_data(15) xor tx_data(14) xor

tx_data(13) xor tx_data(11) xor tx_data(10) xor tx_data(9) xor tx_data(5)
xor tx_data(3) xor

tx_data(1) xor tx_data(0) xor s_tx_address(19) xor s_tx_address(17) xor
s_tx_address(16) xor s_tx_address(15) xor

s_tx_address(11) xor s_tx_address(9) xor s_tx_address(7) xor
s_tx_address(6) xor s_tx_address(4);
                                                        tx_crc(4) <=
tx_cmd(3) xor tx_cmd(2) xor tx_cmd(1) xor tx_cmd(0) xor tx_data(24) xor
```

```
tx_data(21) xor tx_data(18) xor tx_data(17) xor tx_data(16) xor
tx_data(15) xor tx_data(14) xor

tx_data(12) xor tx_data(11) xor tx_data(10) xor tx_data(6) xor tx_data(4)
xor tx_data(2) xor tx_data(1) xor

s_tx_address(18) xor s_tx_address(17) xor s_tx_address(16) xor
s_tx_address(12) xor s_tx_address(10) xor s_tx_address(8) xor
s_tx_address(7) xor s_tx_address(5) xor s_tx_address(0);
                                                        tx_state <=
st_tx_crc;
                                                    end if;

                                                when st_tx_data_frame =>

      if(tx_data_byte(tx_bit_count) = '1') then      --data bits
                                                        tx_bit_time <=
tx_one_bit_time;
                                                    else
                                                        tx_bit_time <=
tx_zero_bit_time;
                                                    end if;
                                                    if(tx_bit_count = 7)
then
                                                        tx_bit_count
<= 0;
                                                        tx_crc(0) <=
tx_crc(0) xor tx_data_byte(7) xor tx_data_byte(5) xor tx_data_byte(3) xor
tx_data_byte(2) xor tx_data_byte(0);
                                                        tx_crc(1) <=
tx_crc(1) xor tx_data_byte(7) xor tx_data_byte(6) xor tx_data_byte(5) xor
tx_data_byte(4) xor tx_data_byte(2) xor tx_data_byte(1) xor
tx_data_byte(0);
                                                        tx_crc(2) <=
tx_crc(2) xor tx_data_byte(6) xor tx_data_byte(1) xor tx_data_byte(0);
                                                        tx_crc(3) <=
tx_crc(3) xor tx_data_byte(5) xor tx_data_byte(3) xor tx_data_byte(1) xor
tx_data_byte(0);
                                                        tx_crc(4) <=
tx_crc(4) xor tx_data_byte(6) xor tx_data_byte(4) xor tx_data_byte(2) xor
tx_data_byte(1);


      if(tx_byte_count = (s_tx_data_frame_size - 1)) then
                                                                tx_state
<= st_tx_crc;

      s_tx_calculate_crc <= '1';
                                                        else

      tx_data_byte <=  tx_din_mem;

      s_tx_addr_mem <= s_tx_addr_mem + 1;

      tx_byte_count <= tx_byte_count + 1;
                                                        end if;
                                                    end if;

                                                when st_tx_crc =>
```

```
      if(tx_crc(tx_bit_count) = '1') then       --crc bits
                                                            tx_bit_time <=
tx_one_bit_time;
                                                      else
                                                            tx_bit_time <=
tx_zero_bit_time;
                                                      end if;
                                                      if(tx_bit_count = 4)
then
                                                            tx_bit_count
<= 0;
                                                            tx_state <=
st_tx_end_delay;
                                                      end if;

                                                when st_tx_end_delay =>
                                                            tx_bit_time <=
tx_one_bit_time;              --frame delay
                                                            output_ctrl_en
:= '0';
                                                            tx_bit_count
<= 0;
                                                            tx_state <=
st_tx_end_delay_2;

                                                when st_tx_end_delay_2 =>
                                                            tx_bit_time <=
2 * tx_one_bit_time;                --frame delay

      s_tx_sending_data <= '0';
                                                            tx_bit_count
<= 0;
                                                            tx_state <=
st_tx_end;

                                                when st_tx_end =>
                                                            tx_bit_time <=
tx_zero_bit_time;             --frame delay
                                                            tx_ack <= '0';
                                                            tx_bit_count
<= 0;
                                                            tx_state <=
st_tx_sync;
                                                            per_tx_enable
<= '0';     -- disable transmition

                                                when st_tx_dummy =>
                                                            tx_bit_time <=
tx_zero_bit_time;             --Dummy bit
                                                            tx_bit_count
<= 0;
                                                            tx_state <=
st_tx_crc;

      s_tx_calculate_crc <= '1';

                                                when others =>
```

```
                                        end case;
                                        s_pin_tx_aux <= s_pin_tx_aux
xor '1';

                                    else
            ----------------------------------------
            --TX Bootstrap operation
            ----------------------------------------
                  -- s_pin_tx_aux = BSL_tx
                  -- pin_crank_out = RST/NMI

                                        tx_bit_time <=
tx_bootst_bit_time;                  --bootstrap bit time (9600bps)
                                        case boot_state is
                                              when boot_start =>
                              --Start Bootstrap
                                                    output_ctrl_en :=
'1';
                                                    s_tx_sending_data <=
'1';
                                                    wait_counter <=
wait_counter + 1;
                                                    if(wait_counter <
20) then                        --Turn off peripheral (wait for total
discharge)
                                                          s_pin_tx_aux
<= '0';
                                                          pin_crank_out
<= '0';

      boot_bit_counter <= 0;
                                                    elsif(wait_counter <
40) then                  --Turn on peripheral
                                                          s_pin_tx_aux
<= '1';
                                                          pin_crank_out
<= '0';
                                                    elsif(wait_counter <
51) then                  --Bootstrap sequence

      boot_bit_counter <= boot_bit_counter + 1;
                                                          s_pin_tx_aux
<= boot_start_vector_a(boot_bit_counter);
                                                          pin_crank_out
<= boot_start_vector_b(boot_bit_counter);
                                                    elsif(wait_counter =
70) then                  --Clean variables. Ready

      boot_data_parity <= '0';
                                                          s_tx_addr_mem
<= (others =>'0');
                                                          boot_data <=
x"00";
                                                          boot_ckl <=
(others =>'0');
                                                          boot_ckh <=
(others =>'0');

      boot_bit_counter <= 0;
```

```
      boot_byte_counter <= 0;
                                                              boot_ck_bit <=
'0';

      boot_sync_en_bit <= '1';

      boot_frame_number <= 0;
                                                              boot_state <=
boot_idle;
                                                          end if;

                                                      when boot_idle =>
                                  --Idle. Waiting for next transmition...
                                                          s_pin_tx_aux <= '1';
                                                          pin_crank_out <=
'1';
                                                          boot_data <= x"00";
                                                          boot_bit_counter <=
0;
                                                          boot_byte_counter <=
0;
                                                          boot_frame_number <=
boot_frame_number + 1;
                                                          if(tx_address(19) =
'0') then                         --boot_frame_number < tx_address(14 downto
7)) then
                                                              boot_state <=
boot_send_sync;
                                                          else
                                                              boot_state <=
boot_end;
                                                          end if;
                                                          tx_bit_time <= 1;
            --start imediatly next byte

                                                      when boot_send_sync =>
            --Sync byte before every cmd
                                                          boot_byte_counter <=
boot_byte_counter + 1;
                                                          boot_sync_en_bit <=
'1';
                                                          if(boot_byte_counter
= 0) then
                                                              boot_data <=
x"80";
                                                              boot_state <=
boot_send_byte;
                                                          else
                                                              boot_state <=
boot_ack_wait;
                                                              wait_counter
<= 0;
                                                              boot_data <=
x"00";
                                                              boot_ckl <=
(others =>'0');
                                                              boot_ckh <=
(others =>'0');
```

```
                                                        boot_ck_bit <=
'0';
      boot_byte_counter <= 0;
                                                    end if;
                                                when boot_send_cmd =>
            -- Send command
                                                    boot_byte_counter <=
boot_byte_counter + 1;
                                                    boot_sync_en_bit <=
'0';

                                                    case
boot_cmd_frame(boot_frame_number - 1)(0) is     --Configure cmd frame size
                                                        when x"12" =>
                              --Write to Flash
      boot_frame_size <= conv_integer(tx_address(6 downto 0));
      boot_cmd_frame(3)(6)(6 downto 0) <= tx_address(6 downto 0);
                                                        when x"10" =>
                              --Send Password
      boot_frame_size <= 42;
                                                        when others =>
      boot_frame_size <= 10;
                                                    end case;
                                                    if(boot_byte_counter
= 0) then                --Start Byte 0x80h
      boot_data <= x"80";
      boot_state <= boot_send_byte;
      boot_data_flash <= x"00";
      elsif(boot_byte_counter = boot_frame_size) then             --Frame
end
      boot_state <= boot_ack_wait;
      wait_counter <= 0;
                                                        boot_ckl
<= (others =>'0');
                                                        boot_ckh
<= (others =>'0');
      boot_byte_counter <= 0;
      boot_bit_counter <= 0;
      elsif(boot_byte_counter = boot_frame_size - 2) then         --
Checksum High
      boot_data <= not boot_ckl;
```

```
      boot_state <= boot_send_byte;
                                                                  boot_ckh
<= boot_ckh xor boot_data;

      elsif(boot_byte_counter = boot_frame_size - 1) then        --
Checksum Low

      boot_data <= not boot_ckh;

      boot_state <= boot_send_byte;
                                                        else

                --Cmd and Data Bytes

      boot_state <= boot_send_byte;

      boot_ck_bit <= boot_ck_bit xor '1';

      if(boot_ck_bit = '0') then

      boot_ckl <= boot_ckl xor boot_data;
                                                                  else

      boot_ckh <= boot_ckh xor boot_data;
                                                                  end if;


      if(boot_cmd_frame(boot_frame_number - 1)(0) = x"12") then  --Write
to Flash

      if(boot_byte_counter < 8) then

      boot_data <= boot_cmd_frame(boot_frame_number -
1)(boot_byte_counter - 1);

      else

      s_tx_addr_mem <= s_tx_addr_mem + 1;

      boot_data <= tx_din_mem;

      end if;


      elsif(boot_cmd_frame(boot_frame_number - 1)(0) = x"10") then     --
Send Password

      if(boot_byte_counter < 8) then

      boot_data <= boot_cmd_frame(boot_frame_number -
1)(boot_byte_counter - 1);

      else

      boot_data <= x"FF";                  --Password 0xFFh

      end if;
```

```
                                                                else
                        --Other cmd's that don't have Data bytes

      boot_data <= boot_cmd_frame(boot_frame_number -
1)(boot_byte_counter - 1);
                                                                end if;
                                                            end if;
                                                            boot_bit_counter <=
0;
                                                            tx_bit_time <= 1;
            --start imediatly next byte

                                                        when boot_send_byte =>
      --Form byte frame
                                                            boot_bit_counter <=
boot_bit_counter + 1;
                                                            case
boot_bit_counter is
                                                                when 0 =>
                        --Start bit

      s_pin_tx_aux <= '0';

      pin_crank_out <= '1';
                                                                when 9 =>
                        --Parity bit

      s_pin_tx_aux <= boot_data_parity;

      pin_crank_out <= '1';
                                                                when 10 =>
                        --Stop bit

      s_pin_tx_aux <= '1';

      pin_crank_out <= '1';

      if(boot_sync_en_bit = '1') then           --Return to

      boot_state <= boot_send_sync;       --If only sync byte
                                                                else

      boot_state <= boot_send_cmd;        --If command frame
                                                                end if;

      wait_counter <= 0;

      boot_bit_counter <= 0;

      boot_data_parity <= '0';
                                                                when others =>
                        --Data bits

      s_pin_tx_aux <= boot_data(boot_bit_counter - 1);

      pin_crank_out <= '1';

      boot_data_parity <= boot_data_parity xor boot_data(boot_bit_counter
- 1);
```

```
                                                    end case;

                                                when boot_ack_wait =>
                --Delay between frames (acknowlegde wait)
                                                    tx_bit_time <=
50000;                                  -- 1ms
                                                    wait_counter <=
wait_counter + 1;
                                                    if(boot_sync_en_bit
= '1') then

      if(wait_counter = 50) then                --50ms wait time

      boot_state <= boot_send_cmd;
                                                        end if;
                                                    else


      if(wait_counter = 250) then          --250ms wait time

      bootstrap_frame_sent <= '1';

      if(bootstrap_next_frame = '1') then

      boot_state <= boot_idle;
                                                            end if;
                                                        end if;

                                                    end if;

                                                when boot_end =>

            --End bootstrap
                                                    wait_counter <=
wait_counter + 1;
                                                    if(wait_counter =
40) then
                                                        boot_state <=
boot_start;
                                                        per_tx_enable
<= '0';           -- disable transmition

      s_tx_sending_data <= '0';
                                                        output_ctrl_en
:= '0';
                                                    end if;
                                                    s_pin_tx_aux <= '0';
                --Power down peripheral
                                                    pin_crank_out <=
'0';
                                                when others =>
                                            end case;

                                        end if;
                                    else
                                        s_pin_tx_aux <= '1';
      --idle state
                                    end if;
                                end if;

                if(s_tx_calculate_crc = '1') then
```

```
                        s_tx_calculate_crc <= '0';
                        -- polynomial: (0 1 2 3 5)
                        --data width: 49
                        --calculate CRC5
                        tx_crc(0) <= tx_crc(0) xor tx_cmd(3) xor
tx_cmd(2) xor tx_cmd(1) xor s_tx_address(19) xor s_tx_address(18) xor
s_tx_address(17) xor s_tx_address(13) xor s_tx_address(11) xor
                                    s_tx_address(9) xor s_tx_address(8) xor
s_tx_address(6) xor s_tx_address(1) xor s_tx_address(0);
                        tx_crc(1) <= tx_crc(1) xor tx_cmd(1) xor
tx_cmd(0) xor s_tx_address(17) xor s_tx_address(14) xor s_tx_address(13)
xor s_tx_address(12) xor
                                    s_tx_address(11) xor s_tx_address(10) xor
s_tx_address(8) xor s_tx_address(7) xor s_tx_address(6) xor
s_tx_address(2) xor s_tx_address(0);
                        tx_crc(2) <= tx_crc(2) xor tx_cmd(3) xor
s_tx_address(19) xor s_tx_address(17) xor s_tx_address(15) xor
s_tx_address(14) xor s_tx_address(12) xor s_tx_address(7) xor
                                    s_tx_address(6) xor s_tx_address(3) xor
s_tx_address(0);
                        tx_crc(3) <= tx_crc(3) xor tx_cmd(3) xor
tx_cmd(2) xor tx_cmd(1) xor tx_cmd(0) xor s_tx_address(19) xor
s_tx_address(17) xor s_tx_address(16) xor s_tx_address(15) xor
                                    s_tx_address(11) xor s_tx_address(9) xor
s_tx_address(7) xor s_tx_address(6) xor s_tx_address(4);
                        tx_crc(4) <= tx_crc(4) xor tx_cmd(3) xor
tx_cmd(2) xor tx_cmd(1) xor tx_cmd(0) xor s_tx_address(18) xor
s_tx_address(17) xor s_tx_address(16) xor s_tx_address(12) xor
                                    s_tx_address(10) xor s_tx_address(8) xor
s_tx_address(7) xor s_tx_address(5) xor s_tx_address(0);
                  end if;

                  if(tx_start = '1' and per_tx_enable = '0') then
                        per_tx_enable <= '1';   --enable transmition
                        tx_ack <= '1';                  --send
acknowledge.
                        s_pin_tx_aux <= '1';
                        tx_state <= st_tx_sync;
                        tx_counter <= 0;
                        tx_bit_time <= 1;
                  end if;
                  if(bootstrap_en = '1' and per_tx_enable = '0') then
                        per_tx_enable <= '1';               --enable
transmition
                        bootstrap_en_ack <= '1';              --send
acknowledge.
                        s_pin_tx_aux <= '1';
                        boot_state <= boot_start;
                        tx_counter <= 0;
                        tx_bit_time <= 1;
                  end if;

                  if(device_is_bus_master = '1') then
                        if(output_ctrl_en = '1') then             --set
output pins state
                              s_pin_tx <= s_pin_tx_aux;
                        else
                              s_pin_tx <= '1';
                        end if;
```

```
                  else
                        if(output_ctrl_en = '1') then             --set
output pins state
                              s_pin_tx <= not s_pin_tx_aux;
                        else
                              s_pin_tx <= '0';
                        end if;
                  end if;

                  if(bootstrap_en = '0') then
                        bootstrap_en_ack <= '0';
                  end if;



            end if;

            debug_tx(11 downto 10) <= s_pin_tx & '0';
      end process;

      ------------------------------------------------
      ------    End of TX State Machine    -----------
      ------------------------------------------------

      ------------------------------------------------
      ------  Mosfet drive State Machine   -----------
      ------------------------------------------------
      process(rst, clk)
      begin
            if (rst = '1') then
                  pin_tx <= '1';
                  mosfet_a_state <= 121;
            elsif( rising_edge(clk)) then
                  case mosfet_a_state is
                        when 0 to 120 =>
                              pin_tx <= '0';
                              mosfet_a_state <= mosfet_a_state + 1;
                        when 121 =>
                              pin_tx <= '1';
                              if(s_pin_tx = '0') then
                                    if(device_is_bus_master = '1') then
      --Master
                                          mosfet_a_state <= 253;
                                    else
                                          mosfet_a_state <= 251;  --(246
at 20ns)
                                    end if;
                              end if;
                        when 122 to 252 =>
                              pin_tx <= '1';
                              mosfet_a_state <= mosfet_a_state + 1;
                        when 253 =>
                              pin_tx <= '0';
                              if(s_pin_tx = '1') then
                                    if(device_is_bus_master = '1') then
                                          mosfet_a_state <= 119;   --(113
at 20ns)
                                    else
                                          mosfet_a_state <= 121;
```

```
                                    end if;
                                end if;
                            when others =>
                                    mosfet_a_state <= 121;
                    end case;
            end if;
      end process;

      -------------------------------------------------
      ------    Begin of RX State Machine  -----------
      -------------------------------------------------

      process(rst, clk)
      begin
            if (rst = '1')    then
                  rx_timer <= 0;
                  rx_state <= st_rx_sync;
                  rx_bit_time_tolerance <= 40;
                  rx_bit_timeout <= 300;
                  rx_ready <= '0';
                  rx_bit_error <= 0;
                  rx_error_count <= (others => '0');
                  rx_bit_count <= 0;
                  old_pin_rx <= '1';
                  s_rx_cmd <= (others => '0');
                  rx_data <= (others => '0');
                  rx_crc_var(4 downto 0) <= (others => '0');
                  rx_calc_crc(4 downto 0) <= (others => '0');
                  s_frame_complete <= '0';
                  s_rx_calculate_crc <= '0';
                  s_rx_calculate_data_crc <= '0';
                  odd_pins <= 0 ;
                  s_rx_receiving_data <= '0';
                  rx_data_byte_size <= (others => '0');

                  st_tx_mem <= st_idle;
                  s_write_byte_count <= (others => '0');
                  s_write_data_mem <= '0';
                  s_rx_addr_mem <= (others => '0');

            elsif (rising_edge(clk)) then

                  if(rx_ack = '1') then         --received data was read.
                        rx_ready <= '0';
                  end if;

                  rx_timer <= rx_timer + 1;
                  if(rx_timer > rx_bit_timeout) then        --bit time
overflow. reset machine
                        rx_timer <= 0;
                        rx_state <= st_rx_sync;
                        rx_bit_count <= 0;
                        rx_one_bit_time <= 0;
                        rx_zero_bit_time <= 0;
                        --rx_bit_timeout <= 1000;
                        old_pin_rx <= pin_rx;
                        s_rx_receiving_data <= '0';

                  else
```

```
                        rx_bit_time_tolerance <= rx_zero_bit_time / 2;
                        if(pin_rx /= old_pin_rx) then       --pin state
has changed?
                              odd_pins <= odd_pins + 1;
                        end if;

                        if(odd_pins > 1) then         --avoid glitches
                              odd_pins <= 0;
                              old_pin_rx <= pin_rx;
                              if(s_tx_sending_data = '0') then
                                    rx_bit_count <= rx_bit_count + 1;
                                    rx_timer <= 0;
                                    case rx_state is
                                          when st_rx_sync =>
                                                case rx_bit_count is
                                                      when 0 =>
                                                            rx_bit_error
<= 0;

      s_rx_receiving_data <= '1';
                                                            rx_calc_crc(4
downto 0) <= (others => '1');
                                                            rx_crc_var(4
downto 0) <= (others => '1');

      rx_data_byte_size <= (others => '0');
                                                            s_rx_cmd <=
(others => '0');
                                                      when 1 =>

      rx_one_bit_time <= rx_timer;              --'1' bit time
                                                      when 2 =>

      rx_zero_bit_time <= rx_timer;             --'0' bit time
                                                            rx_state <=
st_rx_cmd;
                                                            rx_bit_count
<= 0;
                                                      when others =>
                                                            rx_bit_count
<= 0;
                                                end case;

                                          when st_rx_cmd =>
                                                case rx_bit_count is
                                                      when 0 =>
                                                            if(rx_timer <
(rx_one_bit_time + rx_bit_time_tolerance) and rx_timer > (rx_one_bit_time
- rx_bit_time_tolerance)) then

      s_rx_cmd(0) <= '1';
                                                            elsif
(rx_timer < (rx_zero_bit_time + rx_bit_time_tolerance) and rx_timer >
(rx_zero_bit_time - rx_bit_time_tolerance)) then

      s_rx_cmd(0) <= '0';
                                                            else

      rx_bit_error <= rx_bit_error + 1;
```

```
                                                    end if;
                                                when 1 =>
                                                    if(rx_timer <
(rx_one_bit_time + rx_bit_time_tolerance) and rx_timer > (rx_one_bit_time
- rx_bit_time_tolerance)) then

      s_rx_cmd(1) <= '1';
                                                    elsif
(rx_timer < (rx_zero_bit_time + rx_bit_time_tolerance) and rx_timer >
(rx_zero_bit_time - rx_bit_time_tolerance)) then

      s_rx_cmd(1) <= '0';

      if(s_rx_cmd(0) = '0') then

      rx_state <= st_rx_addr20;

      rx_bit_count <= 0;
                                                        end if;
                                                    else

      rx_bit_error <= rx_bit_error + 1;
                                                    end if;

                                                when 2 =>
                                                    if(rx_timer <
(rx_one_bit_time + rx_bit_time_tolerance) and rx_timer > (rx_one_bit_time
- rx_bit_time_tolerance)) then

      s_rx_cmd(2) <= '1';
                                                    elsif
(rx_timer < (rx_zero_bit_time + rx_bit_time_tolerance) and rx_timer >
(rx_zero_bit_time - rx_bit_time_tolerance)) then

      s_rx_cmd(2) <= '0';
                                                    else

      rx_bit_error <= rx_bit_error + 1;
                                                    end if;
                                                    if(s_rx_cmd(1
downto 0) = CMD_BROADCAST) then
                                                        rx_state
<= st_rx_addr20;

      rx_bit_count <= 0;

      elsif(s_rx_cmd(1 downto 0) = CMD_GIMY_ACCESS) then
                                                        rx_state
<= st_rx_addr8;

      rx_bit_count <= 0;
                                                    end if;

                                                when 3 =>
                                                    if(rx_timer <
(rx_one_bit_time + rx_bit_time_tolerance) and rx_timer > (rx_one_bit_time
- rx_bit_time_tolerance)) then

      s_rx_cmd(3) <= '1';
```

```
                                                                elsif
(rx_timer < (rx_zero_bit_time + rx_bit_time_tolerance) and rx_timer >
(rx_zero_bit_time - rx_bit_time_tolerance)) then

      s_rx_cmd(3) <= '0';
                                                                else

      rx_bit_error <= rx_bit_error + 1;
                                                                end if;
                                                                rx_state <=
st_rx_addr15;
                                                                rx_bit_count
<= 0;

                                                          when others =>
                                                                rx_bit_count
<= 0;
                                                    end case;

                                              when st_rx_addr8 =>
                                                    if(rx_timer <
(rx_one_bit_time + rx_bit_time_tolerance) and rx_timer > (rx_one_bit_time
- rx_bit_time_tolerance)) then

      s_rx_address(rx_bit_count) <= '1';
                                                    elsif (rx_timer <
(rx_zero_bit_time + rx_bit_time_tolerance) and rx_timer >
(rx_zero_bit_time - rx_bit_time_tolerance)) then

      s_rx_address(rx_bit_count) <= '0';
                                                    else
                                                          rx_bit_error <=
rx_bit_error + 1;
                                                    end if;
                                                    if(rx_bit_count = 7) then
                                                          if(s_rx_cmd(2) =
CMD_READ_VALUE) then
                                                                rx_state <=
st_rx_dummy;
                                                          else
                                                                rx_state <=
st_rx_bcdp_data;
                                                          end if;
                                                          s_rx_address(19
downto 8) <= (others => '0');
                                                          rx_bit_count <= 0;
                                                    end if;

                                              when st_rx_addr15 =>
                                                    if(rx_timer <
(rx_one_bit_time + rx_bit_time_tolerance) and rx_timer > (rx_one_bit_time
- rx_bit_time_tolerance)) then

      s_rx_address(rx_bit_count) <= '1';
                                                    elsif (rx_timer <
(rx_zero_bit_time + rx_bit_time_tolerance) and rx_timer >
(rx_zero_bit_time - rx_bit_time_tolerance)) then

      s_rx_address(rx_bit_count) <= '0';
```

```
                                                else
                                                      rx_bit_error <=
rx_bit_error + 1;
                                                end if;
                                                if(rx_bit_count = 14) then
                                                      if((s_rx_cmd(2) =
CMD_RESPONSE_USB) or (s_rx_cmd(3) = CMD_WRITE_VALUE)) then
                                                            rx_state <=
st_rx_bcdp_data;
                                                      else
                                                            rx_state <=
st_rx_dummy;
                                                      end if;
                                                      s_rx_address(19
downto 15) <= (others => '0');
                                                      rx_bit_count <= 0;
                                                end if;

                                          when st_rx_addr20 =>
                                                if(rx_timer <
(rx_one_bit_time + rx_bit_time_tolerance) and rx_timer > (rx_one_bit_time
- rx_bit_time_tolerance)) then

      s_rx_address(rx_bit_count) <= '1';
                                                elsif (rx_timer <
(rx_zero_bit_time + rx_bit_time_tolerance) and rx_timer >
(rx_zero_bit_time - rx_bit_time_tolerance)) then

      s_rx_address(rx_bit_count) <= '0';
                                                else
                                                      rx_bit_error <=
rx_bit_error + 1;
                                                end if;
                                                if(rx_bit_count = 19) then
                                                      if(s_rx_cmd(1 downto
0) = CMD_DATA_FRAME) then
                                                            rx_state <=
st_rx_data_frame;

      rx_data_byte_size <= (s_rx_address(7 downto 4) - 1);
                                                      else
                                                            if(s_rx_cmd(2)
= CMD_READ_VALUE) then
                                                                  rx_state
<= st_rx_dummy;
                                                            else
                                                                  rx_state
<= st_rx_bcdp_data;
                                                            end if;
                                                      end if;
                                                      rx_bit_count <= 0;
                                                      rx_byte_count <= 0;
                                                      s_rx_addr_mem <=
"0000";
                                                end if;

                                          when st_rx_bcdp_data =>
```

```
                                                if(rx_timer <
(rx_one_bit_time + rx_bit_time_tolerance) and rx_timer > (rx_one_bit_time
- rx_bit_time_tolerance)) then

      s_rx_data(rx_bit_count) <= '1';
                                                elsif (rx_timer <
(rx_zero_bit_time + rx_bit_time_tolerance) and rx_timer >
(rx_zero_bit_time - rx_bit_time_tolerance)) then

      s_rx_data(rx_bit_count) <= '0';
                                                else
                                                      rx_bit_error <=
rx_bit_error + 1;
                                                end if;
                                                if(rx_bit_count = 24) then
                                                      rx_state <=
st_rx_crc;
                                                      rx_bit_count <= 0;

                                                      -- calculate CRC5
                                                      rx_calc_crc(0) <=
s_rx_data(22) xor s_rx_data(19) xor s_rx_data(18) xor s_rx_data(17) xor
s_rx_data(16) xor s_rx_data(15) xor s_rx_data(13) xor

s_rx_data(12) xor s_rx_data(11) xor s_rx_data(7) xor s_rx_data(5) xor
s_rx_data(3) xor s_rx_data(2) xor s_rx_data(0);
                                                      rx_calc_crc(1) <=
s_rx_data(23) xor s_rx_data(22) xor s_rx_data(20) xor s_rx_data(15) xor
s_rx_data(14) xor

s_rx_data(11) xor s_rx_data(8) xor s_rx_data(7) xor s_rx_data(6) xor
s_rx_data(5) xor s_rx_data(4) xor s_rx_data(2) xor s_rx_data(1) xor
s_rx_data(0);
                                                      rx_calc_crc(2) <=
s_rx_data(24) xor s_rx_data(23) xor s_rx_data(22) xor

s_rx_data(21) xor s_rx_data(19) xor s_rx_data(18) xor s_rx_data(17) xor
s_rx_data(13) xor s_rx_data(11) xor
                                                            s_rx_data(9)
xor s_rx_data(8) xor s_rx_data(6) xor s_rx_data(1) xor s_rx_data(0);
                                                      rx_calc_crc(3) <=
s_rx_data(24) xor s_rx_data(23) xor s_rx_data(20) xor s_rx_data(17) xor
s_rx_data(16) xor s_rx_data(15) xor s_rx_data(14) xor

s_rx_data(13) xor s_rx_data(11) xor s_rx_data(10) xor s_rx_data(9) xor
s_rx_data(5) xor s_rx_data(3) xor
                                                            s_rx_data(1)
xor s_rx_data(0);
                                                      rx_calc_crc(4) <=
s_rx_data(24) xor s_rx_data(21) xor s_rx_data(18) xor s_rx_data(17) xor
s_rx_data(16) xor s_rx_data(15) xor s_rx_data(14) xor

s_rx_data(12) xor s_rx_data(11) xor s_rx_data(10) xor s_rx_data(6) xor
s_rx_data(4) xor s_rx_data(2) xor s_rx_data(1);
                                                end if;

                                          when st_rx_data_frame =>
```

```
                                                if(rx_timer <
(rx_one_bit_time + rx_bit_time_tolerance) and rx_timer > (rx_one_bit_time
- rx_bit_time_tolerance)) then

      rx_data_byte(rx_bit_count) <= '1';
                                                elsif (rx_timer <
(rx_zero_bit_time + rx_bit_time_tolerance) and rx_timer >
(rx_zero_bit_time - rx_bit_time_tolerance)) then

      rx_data_byte(rx_bit_count) <= '0';
                                                else
                                                      rx_bit_error <=
rx_bit_error + 1;
                                                end if;
                                                if(rx_bit_count = 7) then

      s_rx_calculate_data_crc <= '1';
                                                      rx_bit_count <= 0;
                                                      if(rx_byte_count =
rx_data_byte_size) then
                                                            rx_state <=
st_rx_crc;
                                                      end if;
                                                end if;
                                          when st_rx_dummy =>
                                                rx_state <= st_rx_crc;
                                                rx_bit_count <= 0;

                                          when st_rx_crc =>
                                                if(rx_timer <
(rx_one_bit_time + rx_bit_time_tolerance) and rx_timer > (rx_one_bit_time
- rx_bit_time_tolerance)) then

      rx_crc_var(rx_bit_count) <= '1';
                                                elsif (rx_timer <
(rx_zero_bit_time + rx_bit_time_tolerance) and rx_timer >
(rx_zero_bit_time - rx_bit_time_tolerance)) then

      rx_crc_var(rx_bit_count) <= '0';
                                                else
                                                      rx_bit_error <=
rx_bit_error + 1;
                                                end if;

                                                if(rx_bit_count = 4) then
                                                      rx_state <=
st_rx_sync;
                                                      s_rx_calculate_crc
<= '1';
                                                      rx_bit_count <= 0;
                                                end if;
                                          when others =>
                                    end case;
                              end if;
                        end if;
                  end if;
```

```
            ------------------------------------------------------------
-----
            --Validate Data
                  if(s_rx_calculate_data_crc = '1') then
                        s_rx_calculate_data_crc <= '0';
                        rx_calc_crc(0) <= rx_calc_crc(0) xor
rx_data_byte(7) xor rx_data_byte(5) xor rx_data_byte(3) xor
rx_data_byte(2) xor rx_data_byte(0);
                        rx_calc_crc(1) <= rx_calc_crc(1) xor
rx_data_byte(7) xor rx_data_byte(6) xor rx_data_byte(5) xor
rx_data_byte(4) xor rx_data_byte(2) xor rx_data_byte(1) xor
rx_data_byte(0);
                        rx_calc_crc(2) <= rx_calc_crc(2) xor
rx_data_byte(6) xor rx_data_byte(1) xor rx_data_byte(0);
                        rx_calc_crc(3) <= rx_calc_crc(3) xor
rx_data_byte(5) xor rx_data_byte(3) xor rx_data_byte(1) xor
rx_data_byte(0);
                        rx_calc_crc(4) <= rx_calc_crc(4) xor
rx_data_byte(6) xor rx_data_byte(4) xor rx_data_byte(2) xor
rx_data_byte(1);
                        rx_byte_count <= rx_byte_count + 1;
                        s_rx_addr_mem <= s_rx_addr_mem + 1;
                        s_write_data_mem <= '1';
                        debug_rx(7 downto 0) <= rx_data_byte(7 downto 0);
                  end if;

                  if(s_rx_calculate_crc = '1') then
                        s_rx_calculate_crc <= '0';
                        s_frame_complete <= '1';
                        -- calculate CRC5

--                      debug_rx(29 downto 8) <= s_rx_cmd(1) &
s_rx_cmd(0) & s_rx_address(19 downto 0);

                        rx_calc_crc(0) <= rx_calc_crc(0) xor s_rx_cmd(3)
xor s_rx_cmd(2) xor s_rx_cmd(1) xor s_rx_address(19) xor s_rx_address(18)
xor s_rx_address(17) xor s_rx_address(13) xor s_rx_address(11) xor
                              s_rx_address(9) xor s_rx_address(8) xor
s_rx_address(6) xor s_rx_address(1) xor s_rx_address(0);
                        rx_calc_crc(1) <= rx_calc_crc(1) xor s_rx_cmd(1)
xor s_rx_cmd(0) xor s_rx_address(17) xor s_rx_address(14) xor
s_rx_address(13) xor s_rx_address(12) xor
                              s_rx_address(11) xor s_rx_address(10) xor
s_rx_address(8) xor s_rx_address(7) xor s_rx_address(6) xor
s_rx_address(2) xor s_rx_address(0);
                        rx_calc_crc(2) <= rx_calc_crc(2) xor s_rx_cmd(3)
xor s_rx_address(19) xor s_rx_address(17) xor s_rx_address(15) xor
s_rx_address(14) xor s_rx_address(12) xor s_rx_address(7) xor
                              s_rx_address(6) xor s_rx_address(3) xor
s_rx_address(0);
                        rx_calc_crc(3) <= rx_calc_crc(3) xor s_rx_cmd(3)
xor s_rx_cmd(2) xor s_rx_cmd(1) xor s_rx_cmd(0) xor s_rx_address(19) xor
s_rx_address(17) xor s_rx_address(16) xor s_rx_address(15) xor
                              s_rx_address(11) xor s_rx_address(9) xor
s_rx_address(7) xor s_rx_address(6) xor s_rx_address(4);
                        rx_calc_crc(4) <= rx_calc_crc(4) xor s_rx_cmd(3)
xor s_rx_cmd(2) xor s_rx_cmd(1) xor s_rx_cmd(0) xor s_rx_address(18) xor
s_rx_address(17) xor s_rx_address(16) xor s_rx_address(12) xor
```

```
                                    s_rx_address(10) xor s_rx_address(8) xor
s_rx_address(7) xor s_rx_address(5) xor s_rx_address(0);

                  end if;

                  if(s_frame_complete = '1') then

                        s_frame_complete <= '0';
                        s_rx_receiving_data <= '0';

                        --debug_rx(20 downto 16) <= rx_calc_crc(4 downto
0);

                        if(rx_crc_var(4 downto 0) /= rx_calc_crc(4 downto
0)) then
                              rx_bit_error <= 59;
                              rx_error_count <= rx_error_count + 1;
                        elsif(rx_bit_error = 0) then
                              rx_address(19 downto 0) <= s_rx_address(19
downto 0);
                              if(s_rx_cmd(1 downto 0) = CMD_DATA_FRAME)
then
                                    rx_data(24 downto 0) <= x"33" & x"31"
& x"31" & '1';
                              else
                                    rx_data(24 downto 0) <= s_rx_data(24
downto 0);
                              end if;
                              rx_bit_timeout <= 3*rx_one_bit_time;
                              rx_ready <= '1';
                        else
                              rx_error_count <= rx_error_count + 1;
                        end if;
                  end if;

                  --Error counter (to display)
                  if(rx_error_count(3 downto 0) = "1001" ) then
                        rx_error_count(3 downto 0) <= "0000";
                        rx_error_count(7 downto 4) <= rx_error_count(7
downto 4) + 1;
                        if(rx_error_count(7 downto 4) = "1001" ) then
                              rx_error_count(7 downto 4) <= "0000";
                              rx_error_count(11 downto 8) <=
rx_error_count(11 downto 8) + 1;
                              if(rx_error_count(11 downto 8) = "1001" )
then
                                    rx_error_count(11 downto 8) <=
"0000";
                                    rx_error_count(15 downto 12) <=
rx_error_count(15 downto 12) + 1;
                                    if(rx_error_count(15 downto 12) =
"1001" ) then
                                          rx_error_count(3 downto 0) <=
"0000";
                                          rx_error_count(7 downto 4) <=
"0000";
                                          rx_error_count(11 downto 8) <=
"0000";
```

```
                                        rx_error_count(15 downto 12) <=
"0000";
                                    end if;
                                end if;
                            end if;
                        end if;

                        debug_rx(23 downto 8) <= rx_error_count(15 downto 0);


                end if;


                -------------------------------------------------
                -------  Memory interface   ---
                -------------------------------------------------

                        case st_rx_mem is
                            when st_idle =>
                                rx_we_mem <= '0';
                                if(s_write_data_mem = '1') then
                                    rx_we_mem <= '0';
                                    rx_dout_mem <= rx_data_byte;
                                    st_rx_mem <= st_write_rx_mem;
                                end if;

                            when st_write_rx_mem =>
                                    rx_we_mem <= '1';
                                    st_rx_mem <= st_idle;
                            when others =>
                        end case;

        end process;
        ------------------------------------------------
        ------    End of RX State Machine    -----------
        ------------------------------------------------


end Behavioral;
```

1.6 Peripheral Manager

```
----------------------------------------------------------------------------------
---------
-- Company:
-- Engineer:
--
-- Create Date:    15:06:12 04/18/2007
-- Design Name:
-- Module Name:    peripheralManager - Behavioral
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description:
```

```
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------
---------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.NUMERIC_STD.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity peripheralManager is
    Port (
                        clk         : in  std_logic;
                        rst         : in  std_logic;
                        clkdv2      : in  std_logic;

                        pin_rx                  : in std_logic;
                        pin_tx                  : out std_logic;
                        pin_tx_a_n          : out std_logic;
                        pin_debug           : out std_logic;
                        pin_crank_out       : out std_logic;
                        pin_crank_in        : in std_logic;

                        switch : in std_logic_vector(3 downto 0);
                        ------------------------------------------------
-----
                        ------Init of variables to use with LCD debug   -
-----
                        ------------------------------------------------
-----
                        data_out_lcd        : out std_logic_vector(7 downto
0);
                        addr_out_lcd        : out std_logic_vector(4 downto
0);
                        write_out_lcd       : out std_logic;
                        led_out                 : out std_logic_vector(7
downto 0)
                        ------------------------------------------------
-----
                        ------End of variables to use with LCD debug   --
----
                        ------------------------------------------------
-----

                        );
end peripheralManager;

architecture Behavioral of peripheralManager is
```

```
        -- For message CMD field
    constant CMD_DATA_FRAME     : std_logic_vector(1 downto 0) := "00";
    constant CMD_LIVE_PROTOTYPE : std_logic_vector(1 downto 0) := "01";
    constant CMD_BROADCAST      : std_logic_vector(1 downto 0) := "10";
    constant CMD_GIMY_ACCESS    : std_logic_vector(1 downto 0) := "11";

    -- For message "direction" field
    constant CMD_REQUEST_USB  : std_logic := '0';
    constant CMD_RESPONSE_USB : std_logic := '1';

    -- For read/write field
    constant CMD_READ_VALUE   : std_logic := '0';
    constant CMD_WRITE_VALUE  : std_logic := '1';

       --For peripheral interface signal fields
       constant IN_TX_START                          : integer := 0;
       constant IN_RX_ACK                            : integer := 1;
       constant IN_BOOTSTRAP_MODE                 : integer := 2;
       constant IN_BOOTSTRAP_EN                   : integer := 3;
       constant IN_BOOTSTRAP_NEXT_FRAME : integer := 4;

       constant OUT_TX_ACK                           : integer := 0;
       constant OUT_RX_READY                      : integer := 1;
       constant OUT_BOOTSTRAP_EN_ACK       : integer := 2;
       constant OUT_BOOTSTRAP_FRAME_SENT: integer := 3;
       constant OUT_RX_RECEIVING_DATA      : integer := 4;
       constant OUT_TX_SENDING_DATA        : integer := 5;


            signal per_interface_in                    : std_logic_vector(4
downto 0);              --: in std_logic_vector(3 downto 0);

            signal per_interface_out                   : std_logic_vector(5
downto 0);              --: out std_logic_vector(4 downto 0)

            signal per_interface_bus_speed     : std_logic_vector(7
downto 0);         --: in std_logic_vector(7 downto 0);
            signal device_is_bus_master        : std_logic;
      ------------------------------------------------------
      ------Init of variables to use with LCD debug   ------
      ------------------------------------------------------
            type lcdRamType is array (0 to 31) of std_logic_vector(7
downto 0);
            signal lcdRam : lcdRamType;

            type stateLcd is(state0_lcd, state1_lcd,state2_lcd );
            signal current_state_lcd,next_state_lcd : stateLcd;

            type stateTx is(  st_idle, st_tx_1, st_tx2 );
            signal s_st_tx : stateTx := st_idle;

            signal addr_inc_lcd  : std_logic_vector(4 downto 0);
            signal count_lcd       :std_logic_vector(30 downto 0);
            signal aux_char        : std_logic_vector(7 downto 0);

            -- Text to apear after a reset
            constant linea1: string(1 to 16)    := ">>          T0000";
            constant linea2: string(1 to 16)    := ">>          R0000";
            constant linea3: string(1 to 8)     := "LDGADFBC";
```

```
      ------------------------------------------------------
      ------End of variables to use with LCD debug   ------
      ------------------------------------------------------


            signal s_tx_data_man          : std_logic_vector(24 downto
0);
            signal s_tx_address_man       : std_logic_vector(19 downto
0);
            signal s_tx_cmd               : std_logic_vector(3 downto 0);
            signal s_tx_data_msg_size : std_logic_vector(3 downto 0);
            signal s_tx_data_frame_size : std_logic_vector(10 downto 0);
            signal s_tx_data_frame_size_man : std_logic_vector(3 downto
0);

            signal s_rx_data_man          : std_logic_vector(24 downto
0);
            signal s_rx_address_man       : std_logic_vector(19 downto
0);
            signal s_rx_cmd               : std_logic_vector(3 downto 0);

            signal debug_tx               : std_logic_vector(15 downto
0);
            signal debug_rx               : std_logic_vector(29 downto
0);

            signal s_rx_receiving_data : std_logic;
            signal s_tx_sending_data      : std_logic;


            signal msg_inc : natural range 0 to 4000;

            signal boot_frame_number : integer range 0 to 512;
      ------------------------------------------------------
      ------MEM interface
         ------
      ------------------------------------------------------
            type TX_MEM_ST_type is(st_idle,
st_write_tx_mem_0,st_write_tx_mem_1 );
            signal st_tx_mem : TX_MEM_ST_type;

            type RX_MEM_ST_type is(st_idle, st_read_rx_mem );
            signal st_rx_mem : RX_MEM_ST_type;

            signal s_addr_dataFlow_tx_mem : std_logic_vector(3 downto 0);
            signal s_din_dataFlow_tx_mem  : std_logic_vector(7 downto 0);
            signal s_we_dataFlow_tx_mem   : std_logic;

            signal s_addr_perCtrl_tx_mem  : std_logic_vector(3 downto 0);
            signal s_dout_perCtrl_tx_mem  : std_logic_vector(7 downto 0);

            signal s_addr_perCtrl_rx_mem : std_logic_vector(3 downto 0);
            signal s_din_perCtrl_rx_mem  : std_logic_vector(7 downto 0);
            signal s_we_perCtrl_rx_mem   : std_logic;

            signal s_addr_dataFlow_rx_mem  : std_logic_vector(3 downto
0);
            signal s_dout_dataFlow_rx_mem  : std_logic_vector(7 downto
0);
```

```
            signal s_read_data_mem        : std_logic;
            signal s_read_byte_count    : std_logic_vector(3 downto 0);
            signal s_write_data_mem       : std_logic;
            signal s_write_byte_count    : std_logic_vector(3 downto 0);

            signal s_incremental_counter : std_logic_vector(3 downto 0);
            signal s_incremental_counter_2 : std_logic_vector(3 downto
0);


begin


            perController : entity work.peripheralController(Behavioral)
             port map(
                        clk => clk,--: in  std_logic;
                        rst => rst,--: in  std_logic;

                        pin_rx            => pin_rx,--: in std_logic;
                        pin_tx            => pin_tx,--: out std_logic;
                        pin_crank_in      => pin_crank_in,--: in
std_logic;
                        pin_crank_out     => pin_crank_out,--: out
std_logic;


                        per_interface_in             => per_interface_in,
                              --    : in std_logic_vector(3 downto 0);

                        per_interface_out            =>
per_interface_out,                        --    : out std_logic_vector(4
downto 0);
                        per_interface_bus_speed =>
per_interface_bus_speed,          -- : in std_logic_vector(7 downto 0);

                        device_is_bus_master => device_is_bus_master,

            tx_cmd      => s_tx_cmd,
                        tx_data     => s_tx_data_man,    --: in
std_logic_vector(24 downto 0);
                        tx_address  => s_tx_address_man, --: in
std_logic_vector(19 downto 0);

                        tx_data_msg_size   => s_tx_data_msg_size,--: in
std_logic_vector(3 downto 0);
            tx_data_frame_size => s_tx_data_frame_size,

            rx_cmd      => s_rx_cmd,
                        rx_data     => s_rx_data_man,    --: out
std_logic_vector(24 downto 0);
                        rx_address  => s_rx_address_man, --: out
std_logic_vector(19 downto 0);

                  ------------------------------------
                  ------------ Data Memory Interface
                  ------------------------------------
            tx_addr_mem => s_addr_perCtrl_tx_mem, --: out
std_logic_vector(6 downto 0);
```

```
            tx_din_mem  => s_dout_perCtrl_tx_mem, --: in
std_logic_vector(7 downto 0);

            rx_addr_mem => s_addr_perCtrl_rx_mem,--: out
std_logic_vector(3 downto 0),
            rx_we_mem   => s_we_perCtrl_rx_mem,--: out std_logic;
            rx_dout_mem => s_din_perCtrl_rx_mem,--: out
std_logic_vector(7 downto 0)

                        switchper   => switch(0 downto 0),
                        debug_rx    => debug_rx,
                        debug_tx    => debug_tx,
                        led_out           => led_out

                );


   Inst_MemPerCtrlInterfaceTX_boot: entity
work.MemoryPeripheralControlerInterface
      port map (
      clk   => clk,
      a     => s_addr_dataFlow_tx_mem,
      d     => s_din_dataFlow_tx_mem,
      we    => s_we_dataFlow_tx_mem,
      spo   => open,

      dpra  => s_addr_perCtrl_tx_mem,
      dpo   => s_dout_perCtrl_tx_mem
   );


   Inst_MemPerCtrlInterfaceRX_boot: entity
work.MemoryPeripheralControlerInterface
      port map (
      clk   => clk,
      a     => s_addr_perCtrl_rx_mem,
      d     => s_din_perCtrl_rx_mem,
      we    => s_we_perCtrl_rx_mem,
      spo   => open,

      dpra  => s_addr_dataFlow_rx_mem,
      dpo   => s_dout_dataFlow_rx_mem
   );


      s_rx_receiving_data     <=
per_interface_out(OUT_RX_RECEIVING_DATA);
      s_tx_sending_data       <= per_interface_out(OUT_TX_SENDING_DATA);



      ------------------------------------------------
      -------   INIT of the processes to debug LCD   ---
      ------------------------------------------------
      process(rst, clk)
      begin
            if (rst = '1')    then
```

```
                count_lcd <= (others  => '0');
          elsif clk'event and clk='1' then
                count_lcd <= count_lcd + 1;  --  count for delay
          end if;
    end process;


    process(rst, clk, count_lcd(13))
    begin
          if (rst = '1')    then
                current_state_lcd <= state0_lcd;
          elsif rising_edge(clk) then
                if(count_lcd(13 downto 0) = "10000000000000") then
                      current_state_lcd <= next_state_lcd;
                      if (next_state_lcd = state2_lcd  )  then
                            addr_inc_lcd <= addr_inc_lcd + 1;
                      end if;
                end if;
          end if;
    end process;


    process(current_state_lcd, addr_inc_lcd, lcdRam) --, next_state
    begin
          data_out_lcd <= lcdRam(conv_integer(addr_inc_lcd));
          addr_out_lcd <= addr_inc_lcd;
          write_out_lcd <= '0';

          case current_state_lcd is
                when state0_lcd =>
                      data_out_lcd <=
lcdRam(conv_integer(addr_inc_lcd));
                      addr_out_lcd <= addr_inc_lcd;
    next_state_lcd <= state1_lcd;
                when state1_lcd => write_out_lcd <= '1'; next_state_lcd
<= state2_lcd;
                when state2_lcd => write_out_lcd <= '0'; next_state_lcd
<= state0_lcd;
          end case;
    end process;


    process(rst, clk, per_interface_out(OUT_RX_READY), debug_rx,
debug_tx, per_interface_out(OUT_TX_ACK),  s_rx_data_man,
per_interface_out(OUT_BOOTSTRAP_EN_ACK), s_rx_address_man)
    begin
          if (rst = '1')    then
                lcdRam(0) <=
std_logic_vector(to_unsigned(character'pos(linea1(1)), 8));
                lcdRam(1) <=
std_logic_vector(to_unsigned(character'pos(linea1(2)), 8));
                lcdRam(2) <=
std_logic_vector(to_unsigned(character'pos(linea1(3)), 8));
                lcdRam(3) <=
std_logic_vector(to_unsigned(character'pos(linea1(4)), 8));
                lcdRam(4) <=
std_logic_vector(to_unsigned(character'pos(linea1(5)), 8));
                lcdRam(5) <=
std_logic_vector(to_unsigned(character'pos(linea1(6)), 8));
```

```
                    lcdRam(6) <=
std_logic_vector(to_unsigned(character'pos(linea1(7)), 8));
                    lcdRam(7) <=
std_logic_vector(to_unsigned(character'pos(linea1(8)), 8));
                    lcdRam(8) <=
std_logic_vector(to_unsigned(character'pos(linea1(9)), 8));
                    lcdRam(9) <=
std_logic_vector(to_unsigned(character'pos(linea1(10)), 8));
                    lcdRam(10) <=
std_logic_vector(to_unsigned(character'pos(linea1(11)), 8));
                    lcdRam(11) <=
std_logic_vector(to_unsigned(character'pos(linea1(12)), 8));
                    lcdRam(12) <=
std_logic_vector(to_unsigned(character'pos(linea1(13)), 8));
                    lcdRam(13) <=
std_logic_vector(to_unsigned(character'pos(linea1(14)), 8));
                    lcdRam(14) <=
std_logic_vector(to_unsigned(character'pos(linea1(15)), 8));
                    lcdRam(15) <=
std_logic_vector(to_unsigned(character'pos(linea1(16)), 8));
                    lcdRam(16) <=
std_logic_vector(to_unsigned(character'pos(linea2(1)), 8));
                    lcdRam(17) <=
std_logic_vector(to_unsigned(character'pos(linea2(2)), 8));
                    lcdRam(18) <=
std_logic_vector(to_unsigned(character'pos(linea2(3)), 8));
                    lcdRam(19) <=
std_logic_vector(to_unsigned(character'pos(linea2(4)), 8));
                    lcdRam(20) <=
std_logic_vector(to_unsigned(character'pos(linea2(5)), 8));
                    lcdRam(21) <=
std_logic_vector(to_unsigned(character'pos(linea2(6)), 8));
                    lcdRam(22) <=
std_logic_vector(to_unsigned(character'pos(linea2(7)), 8));
                    lcdRam(23) <=
std_logic_vector(to_unsigned(character'pos(linea2(8)), 8));
                    lcdRam(24) <=
std_logic_vector(to_unsigned(character'pos(linea2(9)), 8));
                    lcdRam(25) <=
std_logic_vector(to_unsigned(character'pos(linea2(10)), 8));
                    lcdRam(26) <=
std_logic_vector(to_unsigned(character'pos(linea2(11)), 8));
                    lcdRam(27) <=
std_logic_vector(to_unsigned(character'pos(linea2(12)), 8));
                    lcdRam(28) <=
std_logic_vector(to_unsigned(character'pos(linea2(13)), 8));
                    lcdRam(29) <=
std_logic_vector(to_unsigned(character'pos(linea2(14)), 8));
                    lcdRam(30) <=
std_logic_vector(to_unsigned(character'pos(linea2(15)), 8));
                    lcdRam(31) <=
std_logic_vector(to_unsigned(character'pos(linea2(16)), 8));
                    per_interface_in(IN_TX_START) <= '0';
                    per_interface_in(IN_RX_ACK) <= '0';
                    s_tx_address_man <= (others => '0');
                    s_tx_data_frame_size_man <= (others => '0');
                    msg_inc <= 0;
                    per_interface_in(IN_BOOTSTRAP_MODE) <= '0';
                    st_rx_mem <= st_idle;
```

```
                    st_tx_mem <= st_idle;
                    s_write_byte_count <= (others => '0');
                    s_read_byte_count <= (others => '0');
                    s_incremental_counter <= (others => '0');
                    s_incremental_counter_2 <= (others => '0');
                    boot_frame_number <= 0;
                    per_interface_bus_speed(7 downto 0) <= x"40";
                    device_is_bus_master <= '0';
              elsif rising_edge(clk) then
                          case switch(3 downto 2) is
                                when "00" =>
                                      per_interface_bus_speed(7 downto 0)
<= conv_std_logic_vector(30,8);
                                when "01" =>
                                      per_interface_bus_speed(7 downto 0)
<= conv_std_logic_vector(20,8);
                                when "10" =>
                                      per_interface_bus_speed(7 downto 0)
<= conv_std_logic_vector(15,8);
                                when "11" =>
                                      per_interface_bus_speed(7 downto 0)
<= conv_std_logic_vector(10,8);
                                when others =>
                          end case;

                          case s_st_tx is
                                when st_idle =>
                                      s_st_tx <= st_idle;
                                when st_tx_1 =>
                                      s_st_tx <= st_tx2;
                                when st_tx2 =>
                                      s_st_tx <= st_idle;
                                      s_tx_cmd(3 downto 0) <= '0' &
CMD_WRITE_VALUE & CMD_GIMY_ACCESS;                          -- config
commmand signals
                                      s_tx_address_man <= s_rx_address_man;
                                      s_tx_data_man <= lcdRam(15) &
lcdRam(14) & lcdRam(13) & lcdRam(12)(0);
                                      s_write_data_mem <= '1';
                                      per_interface_in(IN_TX_START) <= '1';
                          end case;

                          if(switch(0) = '0') then
                                device_is_bus_master <= '1';
                                if(switch(1) = '1') then            --
enable bootstrap
                                per_interface_in(IN_BOOTSTRAP_MODE) <= '1';
                                --test
                                s_tx_address_man <= lcdRam(15) & lcdRam(14)
& lcdRam(13)(3 downto 0); --live prototype 11+4  "100000000000" & "1010"
& "0001"; --gimy access
                             s_tx_cmd(3 downto 0) <= lcdRam(15)(3 downto
0);                           -- config commmand signals
                                --end test
                                if(count_lcd(22 downto 0) =
"10000000000000000000000") then
                                      per_interface_in(IN_BOOTSTRAP_EN) <=
'1';
                                      per_interface_in(IN_TX_START) <= '1';
```

```
                                    if(boot_frame_number < 3) then
                                          boot_frame_number <=
boot_frame_number + 1;
                                          s_tx_address_man(19) <= '0';
                                          s_write_data_mem <= '1';
                                    else
                                          s_tx_address_man(19) <= '1';

      per_interface_in(IN_BOOTSTRAP_NEXT_FRAME) <= '1';
                                    end if;
                              end if;

                        else
                              per_interface_in(IN_BOOTSTRAP_MODE) <= '0';
                              if(count_lcd(17 downto 0) =
"10000000000000000") then  -- message send rate 15= 1.3ms between msg 13
=
                                    msg_inc <=  msg_inc + 1;
                                    if(msg_inc = 240) then        --update
data value
                                          lcdRam(15) <= x"3" &
(lcdRam(15)(3 downto 0) + '1');
                                          s_incremental_counter_2 <=
s_incremental_counter_2 + 1;
                                          msg_inc <=  0;
                                    end if;



                                    s_tx_cmd(3 downto 0) <=
CMD_READ_VALUE & CMD_REQUEST_USB & CMD_LIVE_PROTOTYPE;
            -- config commmand signals
                                    --s_tx_cmd(2) <= CMD_REQUEST_USB;
                                    --s_tx_cmd(3) <= CMD_READ_VALUE;
                                    s_tx_data_frame_size_man <= "1010";--
lcdRam(14)(3 downto 0);
                                    s_tx_address_man <= "00000" &
"10000000000" & "0001"; --live prototype 11+4  "100000000000" & "1010" &
"0001"; --gimy access
                                    s_tx_data_man <= lcdRam(15) &
lcdRam(14) & lcdRam(13) & lcdRam(12)(0);
                                    s_write_data_mem <= '1';
                                    per_interface_in(IN_TX_START) <= '1';
                                    if(lcdRam(15) = x"3A") then
                                          lcdRam(15) <= x"30";
                                          lcdRam(14) <= x"3" &
(lcdRam(14)(3 downto 0) + '1');
                                          if(lcdRam(14) = x"3A") then
                                                lcdRam(14) <= x"30";
                                                lcdRam(13) <= x"3" &
(lcdRam(13)(3 downto 0) + '1');
                                                if(lcdRam(13) = x"3A")
then
                                                      lcdRam(13) <= x"30";
                                                      lcdRam(12) <=
"001100" & (lcdRam(12)(1 downto 0) + '1');
                                                      if(lcdRam(12) >=
x"31") then
                                                            lcdRam(12) <=
x"30";
```

```
                                                end if;
                                            end if;
                                        end if;
                                    end if;

                                end if;
                            end if;
                            else
                                device_is_bus_master <= '0';
                            end if;


                            lcdRam(19) <= x"30" + debug_rx(23 downto 20);
                            lcdRam(20) <= x"30" + debug_rx(19 downto 16);
                            lcdRam(21) <= x"30" + debug_rx(15 downto 12);
                            lcdRam(22) <= x"30" + debug_rx(11 downto 8);
--                          lcdRam(24) <= x"30" + debug_rx(20 downto 19);
--                          lcdRam(25) <= x"30" + debug_rx(18 downto 16);

                            if(per_interface_out(OUT_RX_READY) = '1') then
                                per_interface_in(IN_RX_ACK) <= '1';
                                if((s_rx_address_man =
"10000000000000000001") and (s_rx_cmd(1 downto 0) = CMD_BROADCAST)) then
                                    lcdRam(31) <=  "0011000" &
s_rx_data_man(0);
                                    lcdRam(30) <=  s_rx_data_man(8 downto
1);
                                    lcdRam(29) <=  s_rx_data_man(16
downto 9);
                                    lcdRam(28) <=  s_rx_data_man(24
downto 17);
                                    lcdRam(17) <=
std_logic_vector(to_unsigned(character'pos(linea3(7)), 8));
                                    lcdRam(18) <=
std_logic_vector(to_unsigned(character'pos(linea3(8)), 8));
                                elsif(s_rx_cmd(1 downto 0) =
CMD_GIMY_ACCESS) then

                                    if(device_is_bus_master = '1') then
                                        lcdRam(31) <=  "0011" &
s_rx_address_man(3 downto 0);
                                        lcdRam(30) <=  s_rx_data_man(8
downto 1);
                                        lcdRam(29) <=  s_rx_data_man(16
downto 9);
                                        lcdRam(28) <=  s_rx_data_man(24
downto 17);
                                        lcdRam(17) <=
std_logic_vector(to_unsigned(character'pos(linea3(3)), 8));
                                        lcdRam(18) <=
std_logic_vector(to_unsigned(character'pos(linea3(4)), 8));
                                    else
                                        if(s_rx_cmd(2) =
CMD_READ_VALUE) then
                                            s_st_tx <= st_tx_1;
                                        else
                                            lcdRam(31) <=  "0011" &
s_rx_address_man(3 downto 0);
```

```
                                                lcdRam(30) <=
s_rx_data_man(8 downto 1);
                                                lcdRam(29) <=
s_rx_data_man(16 downto 9);
                                                lcdRam(28) <=
s_rx_data_man(24 downto 17);
                                                lcdRam(17) <=
std_logic_vector(to_unsigned(character'pos(linea3(3)), 8));
                                                lcdRam(18) <=
std_logic_vector(to_unsigned(character'pos(linea3(4)), 8));
                                          end if;
                                    end if;
                              elsif(s_rx_cmd(1 downto 0) =
CMD_DATA_FRAME) then
                                    s_read_data_mem <= '1';
                                    lcdRam(17) <=
std_logic_vector(to_unsigned(character'pos(linea3(5)), 8));
                                    lcdRam(18) <=
std_logic_vector(to_unsigned(character'pos(linea3(6)), 8));
                              else
                                    lcdRam(31) <=  "0011" &
s_rx_address_man(3 downto 0);
                                    lcdRam(30) <=  x"21";
                                    lcdRam(29) <=  x"20";
                                    lcdRam(28) <=  x"20";
                                    lcdRam(17) <=
std_logic_vector(to_unsigned(character'pos(linea3(1)), 8));
                                    lcdRam(18) <=
std_logic_vector(to_unsigned(character'pos(linea3(2)), 8));
                              end if;
                        else
                              per_interface_in(IN_RX_ACK) <= '0';
                        end if;

                        if(per_interface_out(OUT_TX_ACK) = '1') then
                              per_interface_in(IN_TX_START) <= '0';
                        end if;

                        if(per_interface_out(OUT_BOOTSTRAP_EN_ACK) = '1')
then
                              per_interface_in(IN_BOOTSTRAP_EN) <= '0';
                        end if;

                  case st_rx_mem is
                        when st_idle =>
                              if(s_read_data_mem = '1') then
                                    s_read_data_mem <= '0';
                                    s_read_byte_count <= "0000";
                                    s_addr_dataFlow_rx_mem <= "0000";
                                    st_rx_mem <= st_read_rx_mem;
                              end if;

                        when st_read_rx_mem =>
                              if(s_read_byte_count = s_rx_address_man(7
downto 4)) then
                                    st_rx_mem <= st_idle;
                              else
                                    s_addr_dataFlow_rx_mem <=
s_addr_dataFlow_rx_mem + 1;
```

```
      lcdRam(conv_integer(s_addr_dataFlow_rx_mem)) <=
s_dout_dataFlow_rx_mem;
                                    s_read_byte_count <=
s_read_byte_count + 1;
                              end if;

                        when others =>
                  end case;

                  case st_tx_mem is
                        when st_idle =>
                              s_we_dataFlow_tx_mem <= '0';
                              per_interface_in(IN_BOOTSTRAP_NEXT_FRAME)
<= '0';
                              if(s_write_data_mem = '1') then
                                    s_write_data_mem <= '0';
                                    s_write_byte_count <= (others =>
'0');
                                    s_incremental_counter <=
s_incremental_counter_2;
                                    st_tx_mem <= st_write_tx_mem_0;
                              end if;

                        when st_write_tx_mem_0 =>
                              if(switch(1) = '1') then            --
bootstrap mode
                                    if(s_write_byte_count =
s_tx_address_man(3 downto 0)) then
                                          st_tx_mem <= st_idle;
                                          --
per_interface_out(OUT_BOOTSTRAP_FRAME_SENT) <= '1';

      per_interface_in(IN_BOOTSTRAP_NEXT_FRAME) <= '1';

                                    else
                                          s_we_dataFlow_tx_mem <= '1';
                                          s_addr_dataFlow_tx_mem <=
s_write_byte_count;
                                          s_din_dataFlow_tx_mem <=
not(s_write_byte_count) & s_incremental_counter;
                                          st_tx_mem <= st_write_tx_mem_1;
                                    end if;
                              else
            -- normal mode
                                    if(s_write_byte_count =
s_tx_address_man(7 downto 4)) then
                                          st_tx_mem <= st_idle;
                                    else
                                          s_we_dataFlow_tx_mem <= '1';
                                          s_addr_dataFlow_tx_mem <=
s_write_byte_count;
                                          s_din_dataFlow_tx_mem <= x"3" &
s_incremental_counter;
                                          st_tx_mem <= st_write_tx_mem_1;
                                    end if;
                              end if;
                        when st_write_tx_mem_1 =>
                              st_tx_mem <= st_write_tx_mem_0;
```

```
                                        s_we_dataFlow_tx_mem <= '0';
                                        s_incremental_counter <=
s_incremental_counter + 1;
                                        s_write_byte_count <=
s_write_byte_count + 1;

                        when others =>
                    end case;
                    end if;


        end process;
        ------------------------------------------------
        -------   END of the processes to debug LCD    ---
        ------------------------------------------------

        ------------------------------------------------
        -------  Memory interface   ---
        ------------------------------------------------


end Behavioral;
```

1.7 Peripheral DataFlow Controller

```
----------------------------------------------------------------------------
---------
-- Company:
-- Engineer:
--
-- Create Date:    14:36:09 12/20/2007
-- Design Name:
-- Module Name:    peripheralDataFlowControl - Behavioral
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------
---------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

use work.ECU2010_package.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
```

```
--library UNISIM;
--use UNISIM.VComponents.all;

entity peripheralDataFlowControl is
      Port(
      clk : in std_logic;
            rst : in std_logic;
      ----------------------------------------------------
      ---- Peripheral Controller interface
      pin_rx : in std_logic;
      pin_tx_a_p : out std_logic;
      pin_crank_out : out std_logic;
      pin_crank_in : in std_logic;


-----------------------------------------------------
-- FIFO OUT interface
            wen_fifo_msgUp    : out std_logic;
      afull_fifo_msgUp : in std_logic;
      dout_fifo_msgUp  : out std_logic_vector(7 downto 0);

-----------------------------------------------------
-- FIFO PER interface
            wen_fifo_per    : out std_logic;
      afull_fifo_per : in std_logic;
      dout_fifo_per  : out std_logic_vector(7 downto 0);

-----------------------------------------------------
-- FIFO IN interface
            ren_fifo_msgIn    : out std_logic;
      empty_fifo_msgIn : in std_logic;
      din_fifo_msg_In  : in std_logic_vector(7 downto 0);

-----------------------------------------------------
-- per2GimyTable interface
      addr_per2GimyTable    : out std_logic_vector(7 downto 0);
      data_in_per2GimyTable : in std_logic_vector(15 downto 0);

-----------------------------------------------------
-- gimy2perTable interface
      data_in_gimy2perTable  : in std_logic_vector (31 downto 0); -- Port
B 8-bit Data Input
      addr_gimy2perTable     : out std_logic_vector(8 downto 0);       --
Port B 1-bit Address Input
--          data_out_gimy2perTable : out std_logic_vector (31 downto 0);
-- Port B 8-bit Data Output
            enable_gimy2perTable   : out std_logic;    -- PortB RAM
Enable Input
      write_gimy2perTable       : out std_logic;    -- Port B Write
Enable Input

-----------------------------------------------------
-- gimy2per Direction bit interface
      addr_gimy2perDirBit    : out std_logic_vector(8 downto 0);
      data_in_gimy2perDirBit : in std_logic_vector(0 downto 0);

-----------------------------------------------------
---- Serial Number Table interface
--      addr_dataFlow_sn_mem : out std_logic_vector(3 downto 0);
```

```
--      dout_dataFlow_sn_mem : out std_logic_vector(31 downto 0);
--      we_dataFlow_sn_mem   : out std_logic;
--      din_dataFlow_sn_mem  : in  std_logic_vector(31 downto 0);

-------------------------------------------------
-- GIMY interface
            data_in_gimy      : in std_logic_vector (31 downto 0);      --
D data
            en_gimy           : out std_logic;
                  -- D EN
            we_gimy           : out std_logic;
                  -- D write enable
            rst_gimy          : out std_logic;
                  -- D reset
            addr_gimy         : out std_logic_vector (9 downto 0);      --
D address
            data_out_gimy  : out std_logic_vector (31 downto 0); -- D
data
            gnt_in_gimy    : in std_logic

      );
end peripheralDataFlowControl;

architecture Behavioral of peripheralDataFlowControl is

   ---------------------------------------------------------
   --
   --
   constant GIMY_ADDRESS     : integer := 15;


   -- For message CMD field
   constant CMD_DATA_FRAME      : std_logic_vector(1 downto 0) := "00";
   constant CMD_LIVE_PROTOTYPE : std_logic_vector(1 downto 0) := "01";
   constant CMD_BROADCAST       : std_logic_vector(1 downto 0) := "10";
   constant CMD_GIMY_ACCESS     : std_logic_vector(1 downto 0) := "11";

   -- For message "direction" field
   constant CMD_REQUEST_USB  : std_logic := '0';
   constant CMD_RESPONSE_USB : std_logic := '1';

   -- For read/write field
   constant CMD_READ_VALUE    : std_logic := '0';
   constant CMD_WRITE_VALUE   : std_logic := '1';


      --For peripheral interface signal fields
      constant IN_TX_START                              : integer := 0;
      constant IN_RX_ACK                                : integer := 1;
      constant IN_BOOTSTRAP_MODE                  : integer := 2;
      constant IN_BOOTSTRAP_EN                    : integer := 3;

      constant OUT_TX_ACK                               : integer := 0;
      constant OUT_RX_READY                       : integer := 1;
      constant OUT_BOOTSTRAP_EN_ACK       : integer := 2;
      constant OUT_RX_RECEIVING_DATA      : integer := 3;
      constant OUT_TX_SENDING_DATA        : integer := 4;
```

```
   -- States of the data flow controler
      type stateDataFlow is(  st_idle,
                              st_wait_transmittion_start,
st_wait_transmittion_end,
                              st_process_cmd,st_process_cmd_extra,


st_write_gimy_viaRoute,st_write_gimy_02,st_write_gimy_03,
                              st_read_gimy_01, st_read_gimy_02,


                              st_send_dst_fifo,
                              st_send_msg_length_msb,
                              st_send_msg_length_lsb,
                              st_send_start_byte,
                              st_send_cmd_msb,
                              st_send_cmd_lsb,
                              st_send_data_id_msb,
                              st_send_data_id_lsb,
                              st_send_data_length_msb,
                              st_send_data_length_lsb,
                              st_send_data,
                              st_send_crc_msb,
                              st_send_crc_lsb,
                              st_send_stop_byte

                              );

   signal s_data_flow_state : stateDataFlow := st_idle;

   signal s_wait_time : std_logic_vector(7 downto 0);


   -- States of the data flow controler
      type stateProcessFifo is( st_receive_cmd, st_process_cmd,
                             st_prc_cmd_plug,
st_prc_cmd_addr_msb,st_prc_cmd_addr_lsb,
                             st_prc_cmd_value_3,
st_prc_cmd_value_2,st_prc_cmd_value_1,
                             st_prc_cmd_value_0
                            );

   signal s_process_fifo_state : stateProcessFifo := st_process_cmd;


   ------------------------------------------------------------
   --- message up interface
   --- start
   signal s_crc_out_actual_value_msgUP : std_logic_vector(15 downto 0);
   signal s_crc_out_old_value_msgUP    : std_logic_vector(15 downto 0);
   signal s_crc_out_new_data_msgUP     : std_logic_vector(7 downto 0);

   -- Registers to configure the message to send to the configuration
manager
   signal s_dst_fifo_reg_msgUP    : std_logic_vector(7 downto 0);
   signal s_msg_length_reg_msgUP  : std_logic_vector(15 downto 0);
   signal s_cmd_to_send_reg_msgUP : std_logic_vector(15 downto 0);
   signal s_data_length_reg_msgUP : std_logic_vector(15 downto 0);
```

```
    signal s_sent_data_counter_msgUP : std_logic_vector(15 downto 0);


    --- stop
    --- message up interface
    ------------------------------------------------------------



    ------------------------------------------------------------
    --- Peripheral Controller interface
    --- start
    signal s_tx_cmd_perCtrl     : std_logic_vector(3 downto 0);
    signal s_tx_data_perCtrl    : std_logic_vector(24 downto 0);
       signal s_tx_address_perCtrl : std_logic_vector(19 downto 0);
    signal s_tx_data_msg_size   : std_logic_vector(3 downto 0);
       signal s_tx_data_frame_size : std_logic_vector(10 downto 0);

--    signal s_tx_cmd     : std_logic_vector(3 downto 0);
--    signal s_tx_data    : std_logic_vector(24 downto 0);
--     signal s_tx_address : std_logic_vector(19 downto 0);
--     signal s_tx_rw_perCtrl   : std_logic;
--     signal s_tx_add_mode : std_logic;
--     signal s_tx_start_perCtrl : std_logic;
--     signal s_tx_ack_perCtrl   : std_logic;

    signal s_addr_dataFlow_tx_mem : std_logic_vector(3 downto 0);
    signal s_din_dataFlow_tx_mem  : std_logic_vector(7 downto 0);
    signal s_we_dataFlow_tx_mem   : std_logic;

    signal s_addr_perCtrl_tx_mem  : std_logic_vector(3 downto 0);
    signal s_dout_perCtrl_tx_mem  : std_logic_vector(7 downto 0);

    signal s_rx_cmd_perCtrl     : std_logic_vector(3 downto 0);
       signal s_rx_data_perCtrl    : std_logic_vector(24 downto 0);
       signal s_rx_address_perCtrl : std_logic_vector(19 downto 0);

    -- internal copy
    signal s_rx_cmd     : std_logic_vector(3 downto 0);
       signal s_rx_data    : std_logic_vector(24 downto 0);
       signal s_rx_address : std_logic_vector(19 downto 0);

--     signal s_rx_rw_perCtrl       : std_logic;
--     signal s_rx_add_mode_perCtrl : std_logic;
--     signal s_rx_ready_perCtrl    : std_logic;
--     signal s_rx_ack_perCtrl      : std_logic;

    signal s_addr_perCtrl_rx_mem : std_logic_vector(3 downto 0);
    signal s_din_perCtrl_rx_mem  : std_logic_vector(7 downto 0);
    signal s_we_perCtrl_rx_mem   : std_logic;

    signal s_addr_dataFlow_rx_mem  : std_logic_vector(3 downto 0);
    signal s_dout_dataFlow_rx_mem  : std_logic_vector(7 downto 0);


--     signal s_bootst_mode      : std_logic;
--    signal s_bootstrap_en     : std_logic;
--    signal s_bootstrap_en_ack : std_logic;
```

```
   --- stop
   --- Peripheral Controller interface
   -------------------------------------------------------

   signal s_plug_id : std_logic_vector(3 downto 0);


   signal s_cmd_from_fifo : std_logic_vector(7 downto 0);


   signal s_data_type : std_logic;

      signal s_per_interface_in_perCtrl        : std_logic_vector(3
downto 0);
   signal s_per_interface_out_perCtrl       : std_logic_vector(4 downto
0);
   signal s_per_interface_bus_speed_perCtrl : std_logic_vector(7 downto
0);


begin



   Inst_MemPerCtrlInterfaceTX_boot: entity
work.MemoryPeripheralControlerInterface
      port map (
      clk   => clk,
      a     => s_addr_dataFlow_tx_mem,
      d     => s_din_dataFlow_tx_mem,
      we    => s_we_dataFlow_tx_mem,
      spo   => open,

      dpra  => s_addr_perCtrl_tx_mem,
      dpo   => s_dout_perCtrl_tx_mem
   );


   Inst_MemPerCtrlInterfaceRX_boot: entity
work.MemoryPeripheralControlerInterface
      port map (
      clk   => clk,
      a     => s_addr_perCtrl_rx_mem,
      d     => s_din_perCtrl_rx_mem,
      we    => s_we_perCtrl_rx_mem,
      spo   => open,

      dpra  => s_addr_dataFlow_rx_mem,
      dpo   => s_dout_dataFlow_rx_mem
   );



inst_peripheralControler: entity work.peripheralController
   PORT MAP(

                        clk => clk,--: in  std_logic;
```

```
                        rst => rst,--: in  std_logic;

                        pin_rx    => pin_rx,--: in std_logic;
                        pin_tx    => pin_tx_a_p,--: out std_logic;
                        pin_crank_in  => pin_crank_in,--: in std_logic;
                        pin_crank_out => pin_crank_out,--: out std_logic;

--                      bootst_mode       => s_bootst_mode,     --: in
std_logic;
--                      bootstrap_en      => s_bootstrap_en,    --: in
std_logic;
--                      bootstrap_en_ack  => s_bootstrap_en_ack,--: out
std_logic;

                        switchper => "1001",--: in std_logic_vector(3
downto 0);

            tx_cmd      => s_tx_cmd_perCtrl,
                        tx_data     => s_tx_data_perCtrl,     --: in
std_logic_vector(24 downto 0);
                        tx_address  => s_tx_address_perCtrl, --: in
std_logic_vector(19 downto 0);
                        tx_data_msg_size   => s_tx_data_msg_size,--: in
std_logic_vector(3 downto 0);
            tx_data_frame_size => s_tx_data_frame_size,

            rx_cmd      => s_rx_cmd_perCtrl,
                        rx_data     => s_rx_data_perCtrl,     --: out
std_logic_vector(24 downto 0);
                        rx_address  => s_rx_address_perCtrl, --: out
std_logic_vector(19 downto 0);

            per_interface_in  => s_per_interface_in_perCtrl,
            per_interface_out => s_per_interface_out_perCtrl,
            per_interface_bus_speed => s_per_interface_bus_speed_perCtrl,

            tx_addr_mem => s_addr_dataFlow_tx_mem, --: out
std_logic_vector(3 downto 0);
            tx_din_mem  => s_din_dataFlow_tx_mem, --: in
std_logic_vector(7 downto 0);

            rx_addr_mem => s_addr_perCtrl_rx_mem,--: out
std_logic_vector(3 downto 0),
            rx_we_mem   => s_we_perCtrl_rx_mem,--: out std_logic;
            rx_dout_mem => s_din_perCtrl_rx_mem--: out std_logic_vector(7
downto 0)
                  );


   --////////////////////////////////////////
   -- Data Flow State Machine
      process (rst, clk)
      begin
            if (rst = '1')    then
         s_data_flow_state <= st_idle;

         -- Communication with GIMy
                  data_out_gimy <= (others => '0');
                  en_gimy     <= '0' ;
```

```
                we_gimy     <= '0' ;
                rst_gimy    <= '1' ;
                addr_gimy <= (others => '0');

        -- Communication with gimy to peripheral table
        addr_gimy2perDirBit <= (others => '0');
        addr_gimy2perTable  <= (others => '0');

        wen_fifo_msgUp <= '0';

        ren_fifo_msgIn <= '0';

        s_wait_time <= (others => '0');

           elsif rising_edge(clk) then
                rst_gimy    <= '1';
        en_gimy     <= '0';
        wen_fifo_msgUp <= '0';

        ren_fifo_msgIn <= '0';

                -- Global State Machine
                case s_data_flow_state is

           when st_wait_transmittion_start =>
              if (s_per_interface_out_perCtrl(OUT_TX_ACK) = '0') then
                 s_data_flow_state <= st_wait_transmittion_start;
              else
                 s_per_interface_in_perCtrl(IN_TX_START) <= '0';

                 if (s_per_interface_out_perCtrl(OUT_TX_SENDING_DATA) =
'0') then
                    -- ERROR sending
                    s_data_flow_state <= st_idle;
                 else
                    s_data_flow_state <= st_wait_transmittion_end;
                 end if;
              end if;

           when st_wait_transmittion_end =>
              if (s_per_interface_out_perCtrl(OUT_TX_SENDING_DATA) =
'1') then
                 s_data_flow_state <= st_wait_transmittion_end;
              else
                 if (s_wait_time = x"00") then
                    -- go to idle
                    s_data_flow_state <= st_idle;
                    -- if we are receiving we wait...
                 else
                    if (s_per_interface_out_perCtrl(OUT_RX_READY) = '1')
then

                    else

                    end if;
                    s_wait_time <= s_wait_time - '1';
                 end if;
              end if;
```

```
                    when st_idle =>
                if (s_per_interface_out_perCtrl(OUT_RX_READY) = '1') then
                   -- message received in controler
                   s_rx_cmd <= s_rx_cmd_perCtrl;
                   s_rx_address <= s_rx_address_perCtrl;
                   s_rx_data    <= s_rx_data_perCtrl;

                   if (s_rx_cmd_perCtrl(1 downto 0) = CMD_DATA_FRAME) then
                      -- Data for fifos
                      -- read values from rx "boot" memory

                      --s_addr_dataFlow_rx_mem
                      --s_dout_dataFlow_rx_mem
                   elsif(s_rx_cmd_perCtrl(1 downto 0) = CMD_GIMY_ACCESS)
then
                       if (s_rx_cmd_perCtrl(2) = CMD_READ_VALUE) then
                         -- read using routing table or read by gimy
address
                         en_gimy      <= '1';
                       else
                         en_gimy      <= '1';
                         we_gimy      <= '1';
                         addr_gimy    <= s_rx_address_perCtrl(9 downto 0);
                         data_out_gimy(24 downto 0) <= s_rx_data_perCtrl;
                         addr_per2GimyTable <= s_rx_address_perCtrl(7
downto 0);

                         s_data_flow_state <= st_write_gimy_viaRoute;
                       end if;

                   elsif(s_rx_cmd_perCtrl(1 downto 0) = CMD_BROADCAST)
then
                      if (s_rx_address_perCtrl(conv_integer(s_plug_id)) =
'1') then
                         en_gimy      <= '1';
                         we_gimy      <= '1';
                         addr_gimy    <= "000000" & s_rx_address_perCtrl(3
downto 0);
                         data_out_gimy(24 downto 0) <= s_rx_data_perCtrl;

                         s_data_flow_state <= st_write_gimy_02;
                      else
                         s_data_flow_state <= st_idle;
                      end if;
                   elsif(s_rx_cmd_perCtrl(1 downto 0) =
CMD_LIVE_PROTOTYPE) then
                      if (s_rx_cmd_perCtrl(2) = CMD_REQUEST_USB) then
                         if (s_rx_cmd_perCtrl(3) = CMD_READ_VALUE) then
                          -- Read value from Gimy and send it to the
requesting plug
                            en_gimy      <= '1';
                            we_gimy      <= '0';
                            addr_gimy    <= s_rx_address_perCtrl(9 downto
0);

                            s_data_flow_state <= st_read_gimy_01;
                         else
                            -- Write value to Gimy
```

```
                              en_gimy      <= '1';
                              we_gimy      <= '1';
                              addr_gimy    <= s_rx_address_perCtrl(9 downto
0);
                              data_out_gimy(24 downto 0) <=
s_rx_data_perCtrl;

                              s_data_flow_state <= st_write_gimy_02;
                           end if;
                        else
                           if (s_rx_cmd_perCtrl(3) = CMD_READ_VALUE) then
                              s_dst_fifo_reg_msgUP    <= "00000000";-- &
s_usb_received_cmd(14 downto 12);--x"06";
                              s_msg_length_reg_msgUP  <= x"000A";
                              s_cmd_to_send_reg_msgUP <= x"0099" or x"8000";
                              s_data_length_reg_msgUP <= x"0000";

                              --s_data_flow_state <= ;
                           else

                           end if;
                        end if;
                     end if;
                  elsif (s_per_interface_out_perCtrl(OUT_RX_RECEIVING_DATA)
= '1') then
                     s_data_flow_state <= st_idle;
                  elsif (s_per_interface_out_perCtrl(OUT_TX_SENDING_DATA) =
'1') then
                     s_data_flow_state <= st_idle;
                  elsif(empty_fifo_msgIn = '0') then
                     -- Values in FIFO
                     ren_fifo_msgIn <= '1';
                     s_data_flow_state <= st_process_cmd_extra;
                  else
                     -- read next value from table
                     s_data_flow_state <= st_idle;
                  end if;

               ------------------------------------------------
               -- Process cmd in message from fifo in ("usb")
               when st_process_cmd_extra =>
                  ren_fifo_msgIn <= '1';
                  s_data_flow_state <= st_process_cmd;
                  s_process_fifo_state <= st_receive_cmd;

               when st_process_cmd =>
                  case s_process_fifo_state is
                     when st_receive_cmd =>
                        if(empty_fifo_msgIn = '0') then
                           ren_fifo_msgIn <= '1';
                           s_cmd_from_fifo <= din_fifo_msg_In;
                           s_process_fifo_state <= st_prc_cmd_plug;
                        else

                        end if;
                     when st_prc_cmd_plug =>
                        if(empty_fifo_msgIn = '0') then
                           ren_fifo_msgIn <= '1';
```

```
                        s_tx_address_perCtrl(19 downto 16) <=
din_fifo_msg_In(3 downto 0);

                        s_process_fifo_state <= st_prc_cmd_addr_msb;
                     else

                     end if;
                  when st_prc_cmd_addr_msb =>
                     if(empty_fifo_msgIn = '0') then
                        ren_fifo_msgIn <= '1';
                        s_tx_address_perCtrl(15 downto 8) <=
din_fifo_msg_In;
                        s_process_fifo_state <= st_prc_cmd_addr_lsb;
                     else

                     end if;
                  when st_prc_cmd_addr_lsb =>
                     s_tx_address_perCtrl(7 downto 0) <= din_fifo_msg_In;

                     if (s_cmd_from_fifo(0) = '1') then
                        s_dst_fifo_reg_msgUP    <= "00000000";-- &
s_usb_received_cmd(14 downto 12);--x"06";
                        s_msg_length_reg_msgUP  <= x"000A";
                        s_cmd_to_send_reg_msgUP <= x"0001" or x"8000";
                        s_data_length_reg_msgUP <= x"0000";

                        s_tx_cmd_perCtrl <= CMD_WRITE_VALUE &
CMD_REQUEST_USB & CMD_LIVE_PROTOTYPE;

                        if(empty_fifo_msgIn = '0') then
                           ren_fifo_msgIn <= '1';
                           s_process_fifo_state <= st_prc_cmd_value_3;
                        else

                        end if;

                     elsif (s_cmd_from_fifo(1) = '1') then
                        s_dst_fifo_reg_msgUP    <= "00000000";-- &
s_usb_received_cmd(14 downto 12);--x"06";
                        s_msg_length_reg_msgUP  <= x"000A";
                        s_cmd_to_send_reg_msgUP <= x"0002" or x"8000";
                        s_data_length_reg_msgUP <= x"0000";

                        -- read value from peripheral
                        s_tx_cmd_perCtrl <= CMD_READ_VALUE &
CMD_REQUEST_USB & CMD_LIVE_PROTOTYPE;

                        s_data_flow_state <= st_send_dst_fifo;

                     else

                     end if;

                  when st_prc_cmd_value_3 =>
                     if(empty_fifo_msgIn = '0') then
                        ren_fifo_msgIn <= '1';
                        s_tx_data_perCtrl(24) <= din_fifo_msg_In(0);
                        s_process_fifo_state <= st_prc_cmd_value_2;
                     else
```

```
                    end if;
                when st_prc_cmd_value_2 =>
                    if(empty_fifo_msgIn = '0') then
                        ren_fifo_msgIn <= '1';
                        s_tx_data_perCtrl(23 downto 16) <=
din_fifo_msg_In;
                        s_process_fifo_state <= st_prc_cmd_value_1;
                    else

                    end if;
                when st_prc_cmd_value_1 =>
                    if(empty_fifo_msgIn = '0') then
                        ren_fifo_msgIn <= '1';
                        s_tx_data_perCtrl(15 downto 8) <=
din_fifo_msg_In;
                        s_process_fifo_state <= st_prc_cmd_value_0;
                    else

                    end if;
                when st_prc_cmd_value_0 =>

                    s_tx_data_perCtrl(7 downto 0) <= din_fifo_msg_In;
                    s_data_flow_state <= st_send_dst_fifo;

                when others =>
                    s_data_flow_state <= st_idle;

                end case;
            when st_write_gimy_viaRoute =>
                en_gimy      <= '1';
                we_gimy      <= '1';
                addr_gimy    <= data_in_per2GimyTable(9 downto 0);
                data_out_gimy(24 downto 0) <= s_rx_data;

                s_data_flow_state <= st_write_gimy_02;

            when st_write_gimy_02 =>
                    en_gimy      <= '1';
                    we_gimy      <= '1';
                    s_data_flow_state <= st_write_gimy_03;

            when st_write_gimy_03 =>
                if (gnt_in_gimy = '1') then
                    en_gimy      <= '0';
                    we_gimy      <= '0';
                    s_data_flow_state <= st_idle;
                else
                    en_gimy      <= '1';
                    we_gimy      <= '1';
                    s_data_flow_state <= st_write_gimy_03;
                end if;


            ------------------------------------------------
            -- read gimy value
            when st_read_gimy_01 =>
                en_gimy      <= '1';
                we_gimy      <= '0';
```

```
                  if (gnt_in_gimy = '1') then
                     s_data_flow_state <= st_read_gimy_02;
                  else
                     s_data_flow_state <= st_read_gimy_01;
                  end if;
               when st_read_gimy_02 =>
                  if (gnt_in_gimy = '1') then
                     if (s_rx_cmd(1) <= CMD_LIVE_PROTOTYPE(1)) then
                        -- send data to requesting plug
                        s_tx_cmd_perCtrl     <= CMD_READ_VALUE &
CMD_RESPONSE_USB & CMD_LIVE_PROTOTYPE;
                        s_tx_address_perCtrl <= s_rx_address(19 downto 0);
                        s_tx_data_perCtrl    <= data_in_gimy(24 downto 0);

                        s_per_interface_in_perCtrl(IN_TX_START) <= '1';
                        --s_tx_start_perCtrl   <= '1';
                     else
                        en_gimy      <= '1';
                        we_gimy      <= '0';
                        s_data_flow_state <= st_idle;
                     end if;
                  else

                     s_data_flow_state <= st_read_gimy_01;
                  end if;

               ------------------------------------------------------------
               -------
               --   Send data
               when st_send_dst_fifo =>
                  if (afull_fifo_msgUp = '0') then
                     wen_fifo_msgUp <= '1';
                     dout_fifo_msgUp <= s_dst_fifo_reg_msgUP;

                     s_data_flow_state <= st_send_msg_length_msb;
                  else
                     s_data_flow_state <= st_send_dst_fifo;
                  end if;
               when st_send_msg_length_msb =>
                  if (afull_fifo_msgUp = '0') then
                     wen_fifo_msgUp <= '1';
                     dout_fifo_msgUp <= s_msg_length_reg_msgUP(15 downto 8);

                     s_data_flow_state <= st_send_msg_length_lsb;
                  else
                     s_data_flow_state <= st_send_msg_length_msb;
                  end if;
               when st_send_msg_length_lsb =>
                  if (afull_fifo_msgUp = '0') then
                     wen_fifo_msgUp <= '1';
                     dout_fifo_msgUp <= s_msg_length_reg_msgUP(7 downto 0);

                     s_data_flow_state <= st_send_start_byte;
                  else
                     s_data_flow_state <= st_send_msg_length_lsb;
                  end if;
               when st_send_start_byte =>
                  if (afull_fifo_msgUp = '0') then
                     wen_fifo_msgUp <= '1';
```

```
                    dout_fifo_msgUp <= x"2A";

                    s_crc_out_new_data_msgUP  <= (others => '0');
                    s_crc_out_old_value_msgUP <= (others => '0');

                    s_data_flow_state <= st_send_cmd_msb;
                 else
                    s_data_flow_state <= st_send_start_byte;
                 end if;
              when st_send_cmd_msb =>
                 if (afull_fifo_msgUp = '0') then
                    wen_fifo_msgUp <= '1';
                    dout_fifo_msgUp <= s_cmd_to_send_reg_msgUP(15 downto
8);

                    s_crc_out_new_data_msgUP  <= s_cmd_to_send_reg_msgUP(15
downto 8);
                    s_crc_out_old_value_msgUP <=
s_crc_out_actual_value_msgUP;

                    s_data_flow_state <= st_send_cmd_lsb;
                 else
                    s_data_flow_state <= st_send_cmd_msb;
                 end if;
              when st_send_cmd_lsb =>
                 if (afull_fifo_msgUp = '0') then
                    wen_fifo_msgUp <= '1';
                    dout_fifo_msgUp <= s_cmd_to_send_reg_msgUP(7 downto 0);

                    s_crc_out_new_data_msgUP  <= s_cmd_to_send_reg_msgUP(7
downto 0);
                    s_crc_out_old_value_msgUP <=
s_crc_out_actual_value_msgUP;

                    s_data_flow_state <= st_send_data_id_msb;
                 else
                    s_data_flow_state <= st_send_cmd_lsb;
                 end if;
              when st_send_data_id_msb =>
                 if (afull_fifo_msgUp = '0') then
                    wen_fifo_msgUp <= '1';
                    dout_fifo_msgUp <= x"00";

                    s_crc_out_new_data_msgUP  <= x"00";
                    s_crc_out_old_value_msgUP <=
s_crc_out_actual_value_msgUP;

                    s_data_flow_state <= st_send_data_id_lsb;
                 else
                    s_data_flow_state <= st_send_data_id_msb;
                 end if;
              when st_send_data_id_lsb =>
                 if (afull_fifo_msgUp = '0') then
                    wen_fifo_msgUp <= '1';
                    dout_fifo_msgUp <= x"01";

                    s_crc_out_new_data_msgUP  <= x"01";
                    s_crc_out_old_value_msgUP <=
s_crc_out_actual_value_msgUP;
```

```
                    s_data_flow_state <= st_send_data_length_msb;
                else
                    s_data_flow_state <= st_send_data_id_lsb;
                end if;
            when st_send_data_length_msb =>
                if (afull_fifo_msgUp = '0') then
                    wen_fifo_msgUp <= '1';
                    dout_fifo_msgUp <= s_data_length_reg_msgUP(15 downto
8);

                    s_crc_out_new_data_msgUP  <= s_data_length_reg_msgUP(15
downto 8);
                    s_crc_out_old_value_msgUP <=
s_crc_out_actual_value_msgUP;

                    s_data_flow_state <= st_send_data_length_lsb;
                else
                    s_data_flow_state <= st_send_data_length_msb;
                end if;
            when st_send_data_length_lsb =>
                if (afull_fifo_msgUp = '0') then
                    wen_fifo_msgUp <= '1';
                    dout_fifo_msgUp <= s_data_length_reg_msgUP(7 downto 0);

                    s_crc_out_new_data_msgUP  <= s_data_length_reg_msgUP(7
downto 0);
                    s_crc_out_old_value_msgUP <=
s_crc_out_actual_value_msgUP;

                    if (s_data_length_reg_msgUP = x"0000") then
                        s_data_flow_state <= st_send_crc_msb;
                    else
                        s_data_flow_state <= st_send_data;
                    end if;
                else
                    s_data_flow_state <= st_send_data_length_lsb;
                end if;

            when st_send_data =>
                if (afull_fifo_msgUp = '0') then
                    wen_fifo_msgUp <= '1';
                    dout_fifo_msgUp <= x"00";

                    s_crc_out_new_data_msgUP  <= x"00";
                    s_crc_out_old_value_msgUP <=
s_crc_out_actual_value_msgUP;

                    s_data_flow_state <= st_send_crc_msb;
                else
                    s_data_flow_state <= st_send_data;
                end if;
            when st_send_crc_msb =>
                if (afull_fifo_msgUp = '0') then
                    wen_fifo_msgUp <= '1';
                    dout_fifo_msgUp <= s_crc_out_actual_value_msgUP(15
downto 8);

                    s_data_flow_state <= st_send_crc_lsb;
```

```
                else
                    s_data_flow_state <= st_send_crc_msb;
                end if;
            when st_send_crc_lsb =>
                if (afull_fifo_msgUp = '0') then
                    wen_fifo_msgUp <= '1';
                    dout_fifo_msgUp <= s_crc_out_actual_value_msgUP(7
downto 0);

                    s_data_flow_state <= st_send_stop_byte;
                else
                    s_data_flow_state <= st_send_crc_lsb;
                end if;
            when st_send_stop_byte =>
                if (afull_fifo_msgUp = '0') then
                    wen_fifo_msgUp <= '1';
                    dout_fifo_msgUp <= x"00";

                    s_data_flow_state <= st_idle;
                else
                    s_data_flow_state <= st_send_stop_byte;
                end if;



            when others =>
                                s_data_flow_state <= st_idle;
                    end case;
            end if;
        end process;


        Inst_CRC_16_8bits_inter_out_crc: entity work.CRC_16_8bits PORT MAP(
                crc_actual_value => s_crc_out_actual_value_msgUP,
                crc_new_data => s_crc_out_new_data_msgUP,
                crc_old_value => s_crc_out_old_value_msgUP
        );


end Behavioral;
```

1.8 Peripheral Crank

```
----------------------------------------------------------------------------
---------
-- Company:
-- Engineer:
--
-- Create Date:    16:03:46 11/09/2007
-- Design Name:
```

```
-- Module Name:    peripheralCrank - Behavioral
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity peripheralCrank is
    Port (
                        clk : in  std_logic;
                        rst : in  std_logic;

                        pin_debug : out std_logic;
                        pin_crank_in : in std_logic;
                        pin_crank_out : out std_logic
                );
end peripheralCrank;

architecture Behavioral of peripheralCrank is

begin

      process(rst, clk)
      begin
            if(rst = '1') then
                  pin_debug <= '0';
                  pin_crank_out <= '0';
            elsif(rising_edge(clk)) then

            end if;
      end process;
end Behavioral;
```

1.9 Peripheral Bootstrap

```
----------------------------------------------------------------------------------
```

```
-- Company:
-- Engineer:
--
-- Create Date:    16:04:32 11/09/2007
-- Design Name:
-- Module Name:    peripheralBootstrap - Behavioral
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------
---------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity peripheralBootstrap is
    Port (
                        clk : in  std_logic;
                        rst : in  std_logic;

                        pin_rx : in std_logic;
                        pin_tx_a : out std_logic;
                        pin_tx_b : out std_logic;
                        pin_crank_in : out std_logic;
                        pin_crank_out : out std_logic;

                        pin_debug : out std_logic;
                        debug_tx : out std_logic_vector(15 downto 0);
                        debug_rx :out std_logic_vector(20 downto 0)

                );
end peripheralBootstrap;

architecture Behavioral of peripheralBootstrap is

            type output_state_type is (idle, start_bootstrap, send_byte);
            signal output_state : output_state_type;
            signal boot_state : natural range 0 to 15;
            signal timer, bit_time : natural range 0 to 2000;
            signal boot_start_vector_a, boot_start_vector_b :
std_logic_vector(9 downto 0);
            signal boot_data : std_logic_vector(7 downto 0);
begin
      process(rst, clk)
      begin
```

```
            if(rst = '1') then
                  output_state <= idle;
                  boot_state <= 0;
                  timer <= 0;
                  bit_time <= 1000;
                  boot_start_vector_a <= "0010000111";
                  boot_start_vector_b <= "0000001000";
                  boot_data <= x"96";
            elsif(rising_edge(clk)) then
                  timer <= timer + 1;
                  if(timer = bit_time)then
                        timer <= 0;
                        case output_state is
                              when idle =>
                                    pin_tx_a <= '1';
                                    pin_tx_b <= '0';
                                    pin_crank_out <= '1';
                                    boot_state <= 0;
                              when start_bootstrap =>
                                    boot_state <= boot_state + 1;
                                    if(boot_state = 10) then
                                          boot_state <= 0;
                                          output_state <= idle;
                                    end if;
                                    pin_tx_a <=
boot_start_vector_a(boot_state);
                                    pin_crank_out <=
boot_start_vector_b(boot_state);
                              when send_byte =>
                                    boot_state <= boot_state + 1;
                                    if(boot_state = 8) then
                                          output_state <= idle;
                                          boot_state <= 0;
                                    else
                                          pin_tx_a <=
boot_data(boot_state);
                                          pin_crank_out <= '1';
                                    end if;
                              when others =>
                        end case;
                  end if;
            end if;
      end process;
end Behavioral;
```